
2020

Web Almanac

HTTP Archive's annual
state of the web report



Table of Contents

Introduction

Foreword	iii
----------------	-----

Part I. Page Content

Chapter 1: CSS	1
Chapter 2: JavaScript	75
Chapter 3: Markup	105
Chapter 4: Fonts	133
Chapter 5: Media	
Chapter 6: Third Parties	151

Part II. User Experience

Chapter 7: SEO	167
Chapter 8: Accessibility	207
Chapter 9: Performance	231
Chapter 10: Privacy	255
Chapter 11: Security	267
Chapter 12: Mobile Web	305
Chapter 13: Capabilities	327
Chapter 14: PWA	345

Part III. Content Publishing

Chapter 15: CMS	361
Chapter 16: Ecommerce	
Chapter 17: Jamstack	381

Part IV. Content Distribution

Chapter 18: Page Weight	395
Chapter 19: Compression	405

Chapter 20: Caching	415
Chapter 21: Resource Hints	453
Chapter 22: HTTP/2	469

Appendices

Methodology.....	493
Contributors.....	503

Foreword

2020 has been a year many of us would like to forget. It's rare for a community as globalized as ours to be affected by events as far-reaching as the COVID-19 pandemic and protests against racial injustice. These events almost discouraged us from restarting the project this year—with so many people physically and emotionally drained, how could we expect anyone to want to contribute, let alone have the time and energy for it? We proceeded with caution, hoping there was still community interest.

The purpose of this edition of the Web Almanac is not to forget about 2020, but to memorialize it. For better or worse, this is a chapter in our history. Despite all of the external pressures of this year, *over a hundred contributors* from the web community signed up and volunteered countless hours of their time for a project dedicated to remembering 2020 and the state of the web. Amazingly, we actually managed to *expand* the scope of this year's edition by adding three new chapters and only losing one.

When I ask contributors what they enjoy most about the project, the answer is almost always about the people. We work together as teams, we support each other, and in only five months' time we're able to build the equivalent of a 500+ page book! It's an enormous challenge, and while we haven't solved the world's problems, we've shown what's possible when people choose to work together.

Please enjoy the 2020 Web Almanac, the culmination of our labor of love for the web. And be sure to reach out if you'd like to join the team.

— Rick Visconti, *Web Almanac Editor-in-Chief*

Part I Chapter 1

CSS [UNEDITED]



Written by Lea Verou, Chris Lilley, and Rachel Andrew

Reviewed by Estelle Weyl, Elika Etemad aka fantasai, Jens Oliver Meiert, Miriam Suzanne, Catalin Rosu, and Andy Bell

Analyzed by Rick Viscomi, Lea Verou, and Pokidov N. Dmitry

Introduction

Cascading Style Sheets (CSS) is a language used to lay out, format, and paint web pages and other media. It is one of the three main languages for building websites, the other two being HTML, used for structure, and JavaScript, used to specify behavior.

In last year's inaugural Web Almanac, we looked at a variety of CSS metrics measured through 41 SQL queries over the HTTP Archive corpus, to assess the state of the technology in 2019. This year, we went a lot deeper, to measure not only how many pages use a given CSS feature, but also *how* they use it.

Overall, what we observed was a Web in two different gears when it comes to CSS adoption. In our blog posts and twitter bubbles, we tend to mostly discuss the newest and shiniest. However, there are still *millions* of sites using decade-old code. Things like vendor prefixes from a bygone era, proprietary IE filters, and floats for layout, in all their clearfix glory. But we also

observed impressive adoption of many new features, even features that only got support across the board this very year, like `min()` and `max()`. However, there is generally an inverse correlation between how cool something is perceived to be and how much it's actually used, e.g. cutting-edge Houdini features were practically nonexistent.

Similarly, in our conference talks, we often tend to focus on complicated, elaborate use cases that make heads explode and twitter feeds fill with "CSS can do *that*!". However, as it turns out, most CSS usage in the wild is fairly simple. CSS Variables are mostly used as constants and rarely refer to other variables, `calc()` is mostly used with two terms, gradients mostly have two stops and so on.

The Web is not a teenager any more. It is now 30 years old, and acts like it. It tends to favor stability over new bling, and readability over complexity, occasional guilty pleasures aside.

Methodology

The HTTP Archive crawls millions of pages every month and runs them through a private instance of WebPageTest to store key information of every page. (You can learn more about this in our methodology).

For this year, we decided to involve the community in which metrics to study. We started with an app to propose metrics and vote on them. In the end, there were so many interesting metrics that we ended up including nearly all of them. We only excluded Font metrics, since there is a whole separate Fonts chapter and there was significant overlap.

The data in this chapter took 121 SQL queries to produce, totaling over 10K lines of SQL, which includes 3K lines of JS, making it the largest chapter in Web Almanac's history.

A lot of engineering work went into making this scale of analysis feasible. Like last year, we put all CSS code through a CSS parser, and stored the Abstract Syntax Trees (AST) for all stylesheets in the corpus, resulting in a whopping 10 TB of data. This year, we also developed a library of helpers that operate on this AST, and a selector parser, both of which were also released as separate open source projects. Most metrics involved JS to collect data from a single AST, and SQL to aggregate this data over the entire corpus. Curious how your own CSS does against our metrics? We made an online playground where you can try them out on your own sites.

For certain metrics, looking at the CSS AST was not enough. We wanted to look at SCSS wherever it was provided via sourcemaps as it shows us what developers *need* from CSS that is not yet possible, whereas studying CSS shows us what developers currently use that is. For that, we had to use a *custom metric*, i.e. JS code that runs in the crawler when it visits a given

page. We could not use a proper SCSS parser as that could slow down the crawl too much, so we had to resort to regular expressions (*oh, the horror!*). Despite the crude approach, we got a plethora of insights!

Custom metrics were also used for part of the Custom properties analysis. While we can get a lot of information about custom property usage from the stylesheets alone, we cannot build a dependency graph without being able to look at the DOM tree for context, as custom properties are inherited. Looking at the computed style of the DOM nodes also gives us information like what kinds of elements each property is applied to, and which of them are registered, information that we also cannot get from the stylesheets.

Unless otherwise noted, stats presented in this chapter refer to the set of mobile pages.

Usage

While JavaScript far surpasses CSS in its share of page weight, CSS has certainly grown in size over the years, with the median desktop page loading 62 KB of CSS code, and 1 in 10 pages loading more than 240 KB of CSS code. Mobile pages do use slightly less CSS code across all percentiles, but only by 4 to 7 KB. While this is definitely greater than previous years, it doesn't come close to JavaScript's whopping median of 444 KB and top 10% of 1.2 MB

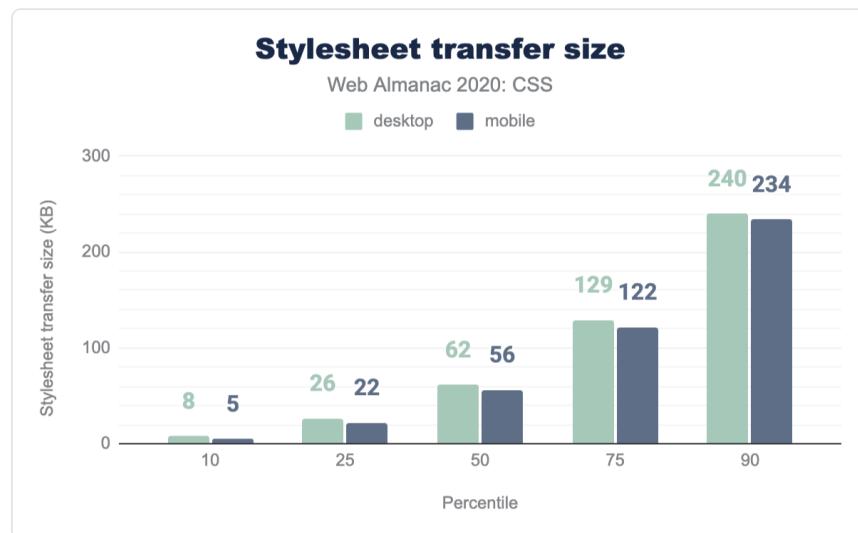


Figure 1.1. Distribution of the stylesheet transfer size per page.

It would be reasonable to assume that a lot of this CSS is generated via preprocessors or other

build tools, however **only about 15%** included sourcemaps. It is unclear whether this says more about sourcemap adoption, or build tool usage. Of those, the overwhelming majority (45%) came from other CSS files, indicating usage of build processes that operate on CSS files, such as minification, autoprefixer, and/or PostCSS. Sass was far more popular than Less (34% of stylesheets with sourcemaps vs 21%), with SCSS being the more popular dialect (33% for .scss vs 1% for .sass).

All these kilobytes of code are typically distributed across multiple files and `<style>` elements; only about 7% of pages concentrate all their CSS code in one remote stylesheet, as we are often taught to do. In fact, the median page contains 3 `<style>` elements and 6 (!) remote stylesheets, with 10% of them carrying over 14 `<style>` elements and over 20 remote CSS files! While this is suboptimal on desktop, it really kills performance on mobile, where round-trip latency is more important than raw download speed.

Shockingly, the maximum number of stylesheets per page is an incredible 26,777 `<style>` elements and 1,379 remote ones! I'd definitely want to avoid loading *that* page!

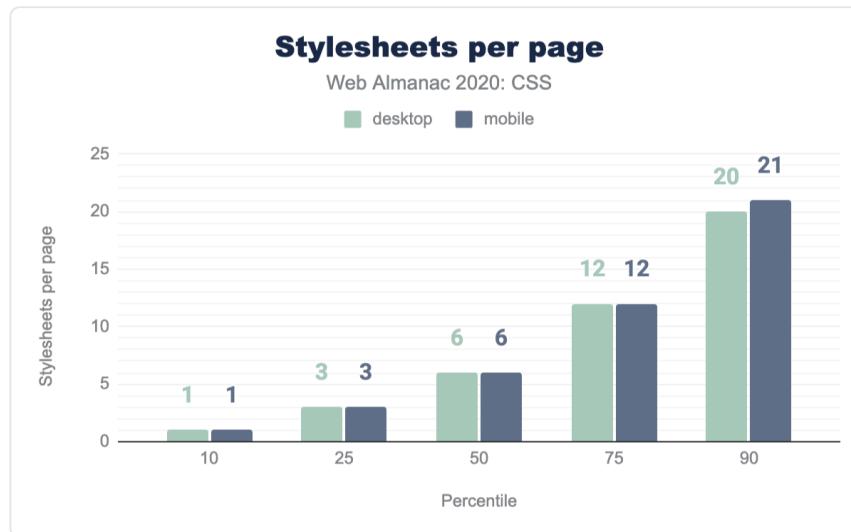


Figure 1.2. Distribution of the number of stylesheets per page.

Another metric of size is the number of rules. The median page carries a total of 448 rules and 5,454 declarations. Interestingly, 10% of pages contain a tiny amount of CSS: fewer than 13 rules! Despite mobile having slightly smaller stylesheets, it also has slightly more rules, indicating smaller rules overall (as it tends to happen with media queries).



Figure 1.3. Distribution of the total number of style rules per page.

Selectors and the cascade

Class names

What do developers use class names for these days? To answer this question, we looked at the most popular class names. The list was dominated by Font Awesome classes, with 192 out of 198 being `fa` or `fa-*`! The only thing that initial exploration could tell us was that Font Awesome is exceedingly popular and is used by almost one third of websites!

However, once we collapsed `fa-*` and then `wp-*` classes (which come from WordPress, another exceedingly popular piece of software), we got more meaningful results. Omitting these, state-related classes seem to be most popular, with `.active` occurring in nearly half of websites, and `.selected` and `.disabled` following soon after.

Only a few of the top classes were presentational, with most of those being either alignment related (`pull-right` and `pull-left` from older Bootstrap, `alignright`, `alignleft` etc) or `clearfix`, which still occurs in 22% of websites, despite floats being superseded as a layout method by the more modern Grid and Flexbox modules.

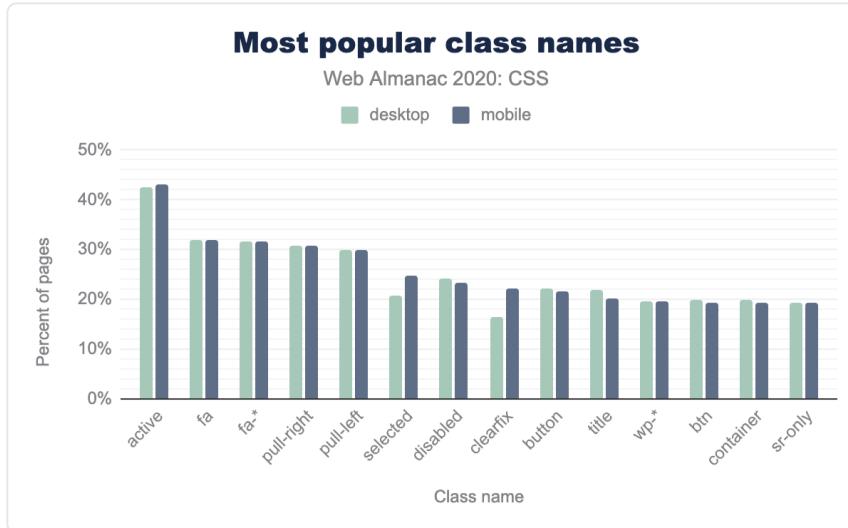


Figure 1.4. The most popular class names by the percent of pages.

IDs

Despite IDs being discouraged these days in some circles due to their much higher specificity, most websites still use them, albeit sparingly. Fewer than half of pages used more than one ID in any of their selectors (had a max specificity of $(1,x,y)$ or less) and nearly all had a median specificity that did not include IDs $(0,x,y)$.

But what are these IDs used for? It turns out that the most popular IDs are structural:

```
#content, #footer, #header, #main, despite corresponding HTML elements.
```

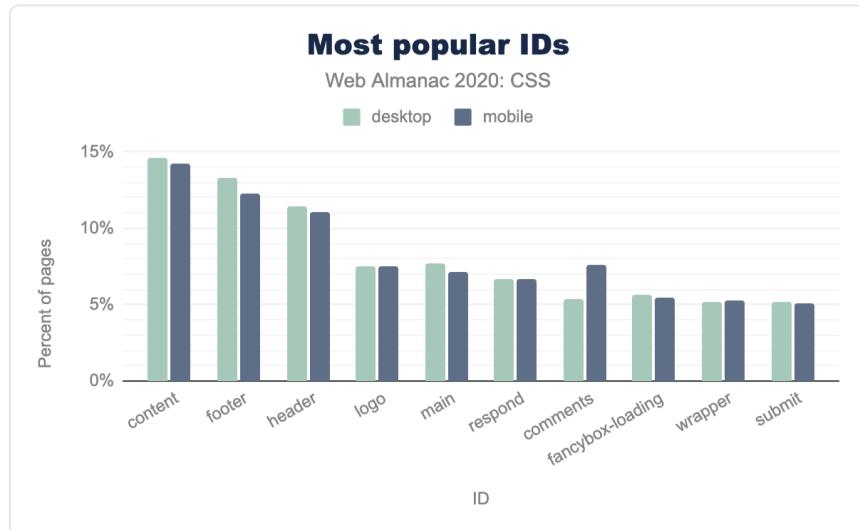


Figure 1.5. The most popular IDs by the percent of pages.

IDs can also be used to intentionally reduce or increase specificity. The specificity hack of writing an ID selector as an attribute selector (`[id="foo"]` instead of `#foo`) was surprisingly rare, with only 0.3% of pages using it at least once. Another ID-related specificity hack, using a negation + descendant selector like `:not (#nonexistent) .foo` instead of `.foo` to increase specificity, was also very rare, appearing in only 0.1% of pages.

!important

Instead, the old, crude `!important` is still used a fair bit despite its well-known drawbacks. The median page uses `!important` in nearly 2% of its declarations, or 1 in 50. Some developers literally cannot get enough of it: we found 2304 desktop pages and 2138 mobile ones that use `!important` in every single declaration!

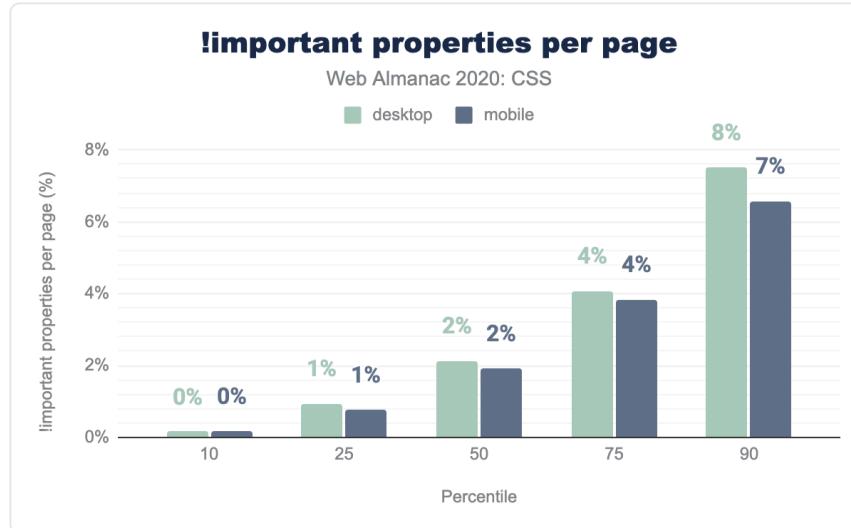


Figure 1.6. Distribution of the percent of `!important` properties per page.

What is it that developers are so keen to override? We looked at breakdown by property, and found that nearly 80% of pages use `!important` with the `display` property. It is a common strategy to apply `display: none !important` to hide content in helper classes to override existing CSS that uses `display` to define a layout mode. This is a side effect of what, in hindsight, was a flaw in CSS. It combined three orthogonal characteristics into one: internal layout mode, flow behavior, and visibility status are all controlled by the `display` property. There are efforts to separate out these values into separate `display` keywords so that they can be tweaked independently via custom properties, but browser support is virtually nonexistent for the time being.

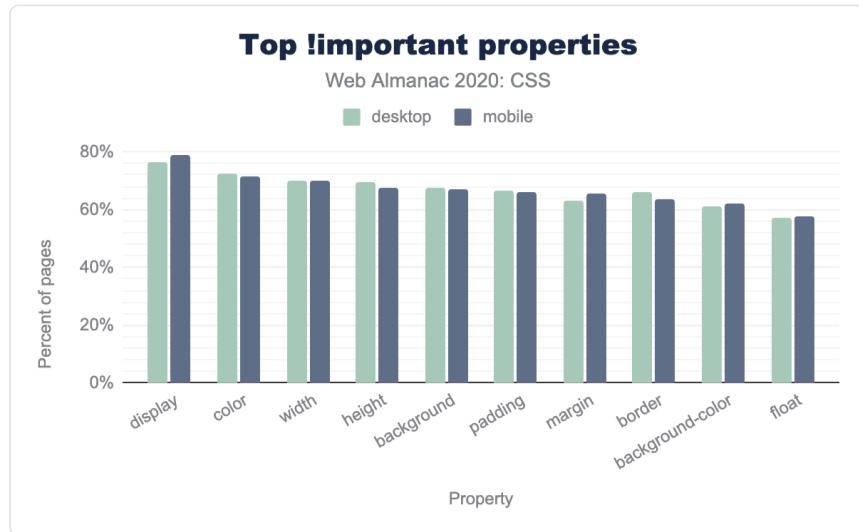


Figure 1.7. The top `!important` properties by the percent of pages.

Specificity and classes

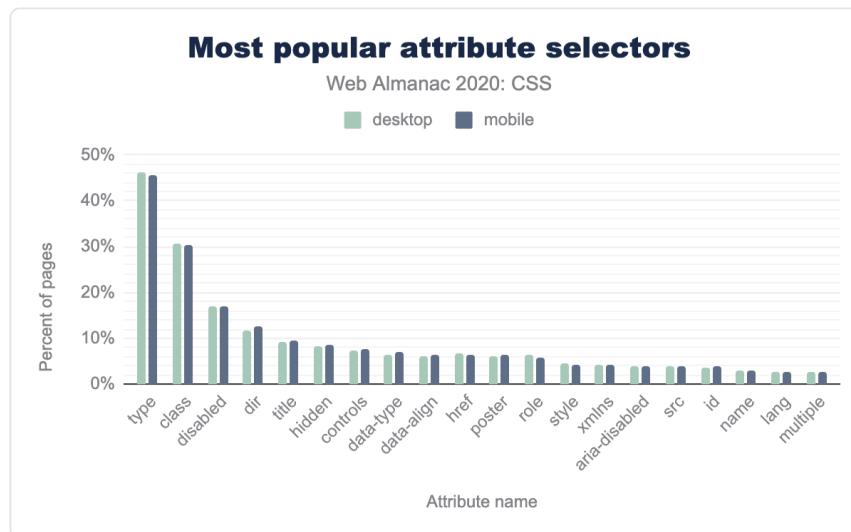
Besides keeping ids and `!important`'s few and far between, there is a trend to circumvent specificity altogether by cramming all the selection criteria of a selector in a single class name, thus forcing all rules to have the same specificity and turning the cascade into a simpler last-one-wins system. BEM is a popular methodology of that type, albeit not the only one. While it is difficult to assess how many websites use BEM-style methodologies exclusively, since following it in every rule is rare (even the BEM website uses multiple classes in many selectors), about 10% of pages had a median specificity of (0,1,0), which may indicate mostly following a BEM-style methodology. On the opposite end of BEM, often developers use duplicated classes to increase specificity and nudge a selector ahead of another one (e.g. `.foo .foo` instead of `.foo`). This kind of specificity hack is actually more popular than BEM, being present in 14% of mobile websites (9% of desktop)! This may indicate that most developers do not actually want to get rid of the cascade altogether, they just need more control over it.

Percentile	Desktop	Mobile
10	0,1,0	0,1,0
25	0,2,0	0,1,2
50	0,2,0	0,2,0
75	0,2,0	0,2,0
90	0,3,0	0,3,0

Figure 1.8. Distribution of the median specificity per page.

Attribute selectors

The most popular attribute selector, by far, is on the `type` attribute, used in 45% of pages, likely to style inputs of different types, e.g. to style textual inputs differently from radios, checkboxes, sliders, file upload controls etc.

*Figure 1.9. The most popular attribute selectors by the percent of pages.*

Pseudo-classes and pseudo-elements

There is always a lot of inertia when we change something in the Web platform after it's long

established. As an example, the Web has still largely not caught up with pseudo-elements having separate syntax compared to pseudo-classes, even though this was a change that happened over a decade ago. All pseudo-elements that are also available with a pseudo-class syntax for legacy reasons are **vastly** more widespread (2.5x to 5x!) with the pseudo-class syntax.

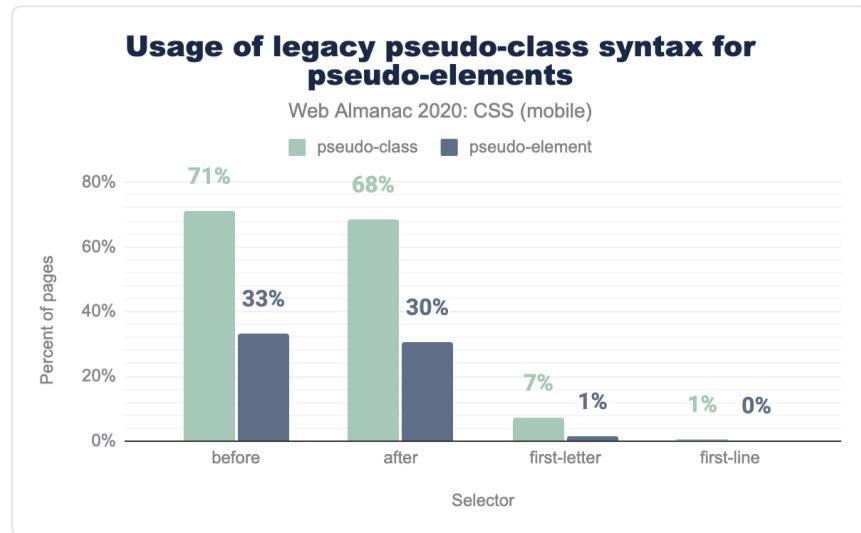


Figure 1.10. Usage of legacy `:pseudo-class` syntax for `::pseudo-elements` as a percent of mobile pages.

By far the most popular pseudo-classes are user action ones, with `:hover`, `:focus`, and `:active` at the top of the list, all used in over two thirds of pages, indicating that developers like the convenience of specifying declarative UI interactions.

`:root` seems far more popular than is justified by its function, used in one third of pages. In HTML content, it just selects the `<html>` element, so why didn't developers just use `html`? A possible answer may lie in a common practice related to defining custom properties, which are also highly used, on the `:root` pseudo-class. Another answer may lie in specificity: `:root`, being a pseudo-class, has a higher specificity than `html : (0, 1, 0)` vs `(0, 0, 1)`. It is a common hack to increase specificity of a selector by prepending it with `:root`, e.g. `:root .foo` has a specificity of `(0, 2, 0)` compared to just `(0, 1, 0)` for `.foo`. This is often all that is needed to nudge a selector slightly over another one in the cascade race and avoid the sledgehammer that is `!important`. To test this hypothesis, we also measured exactly that: how many pages use `:root` at the start of a descendant selector? The results verified our hypothesis: a remarkable 29% of pages use `:root` that way! Furthermore, 14% of desktop pages and 19% of mobile pages use `html` at the start of a descendant selector, possibly to give the selector an even

smaller specificity boost. The popularity of these specificity hacks strongly indicates that developers need more fine grained control to tweak specificity than what is afforded to them via `!important`. Thankfully, this is coming soon with `:where()`, which is already implemented across the board (albeit behind a flag in Chrome for now).

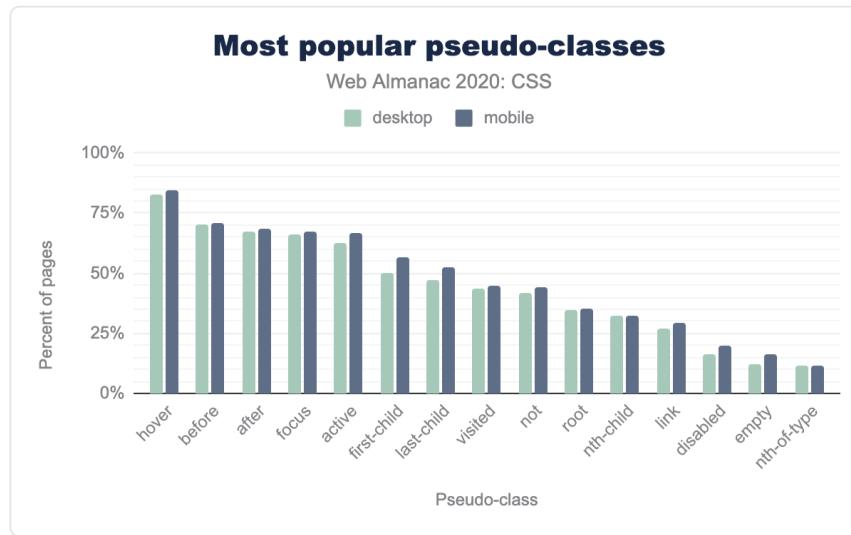


Figure 1.11. The most popular pseudo-classes as a percent of pages.

When it comes to pseudo-elements, after the usual suspects `::before` and `::after`, nearly all popular pseudo-elements were browser extensions for styling form controls and other built-in UI, strongly echoing the developer need for more fine-grained control over styling of built in UI. Styling of focus rings, placeholders, search inputs, spinners, selection, scrollbars, media controls was especially popular.

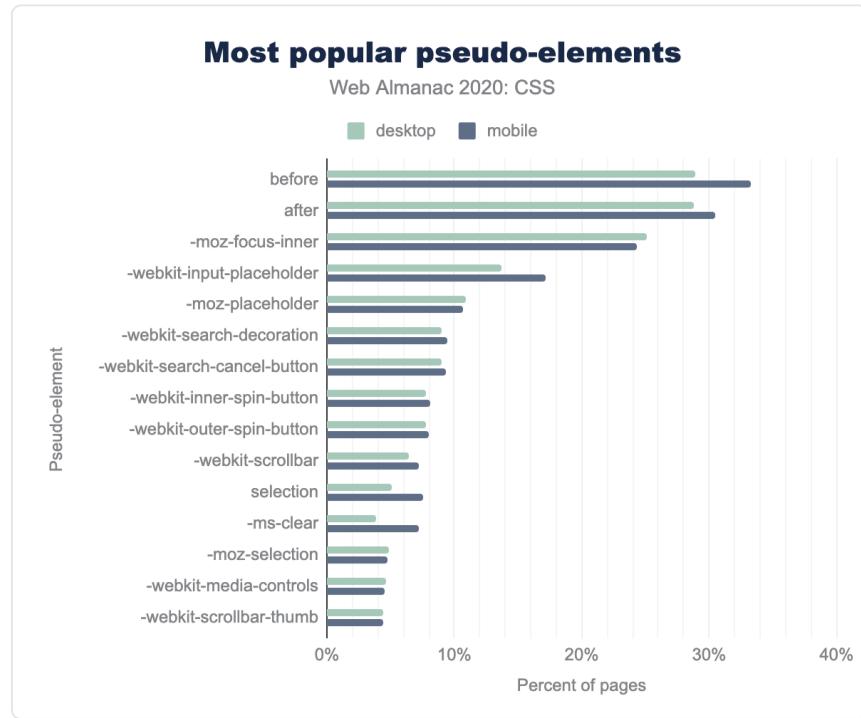


Figure 1.12. The most popular pseudo-elements as a percent of pages.

Values and units

Lengths

The humble `px` unit has gotten a lot of negative press over the years. At first, because it didn't play nicely with old IE's zoom functionality, and, more recently, because there are better units for most tasks that scale based on another design factor, such as viewport size, element font size, or root font size, reducing maintenance effort by making implicit design relationships explicit. The main selling point of `px` – its correspondence to one device pixel giving designers full control – is also gone now, as a pixel is not a device pixel anymore with the modern high pixel density screens. Despite all this, CSS pixels still nearly ubiquitously drive the Web's designs.

72.58%

Figure 1.13. Percentage of `<length>` values that use the `px` unit.

The `px` unit is still going strong as the most popular length unit overall, with a whopping 72.58% of all length values across all stylesheets using `px`! And if we exclude percentages (since they are not really a unit) the share of `px` increases even more, to 84.14%.

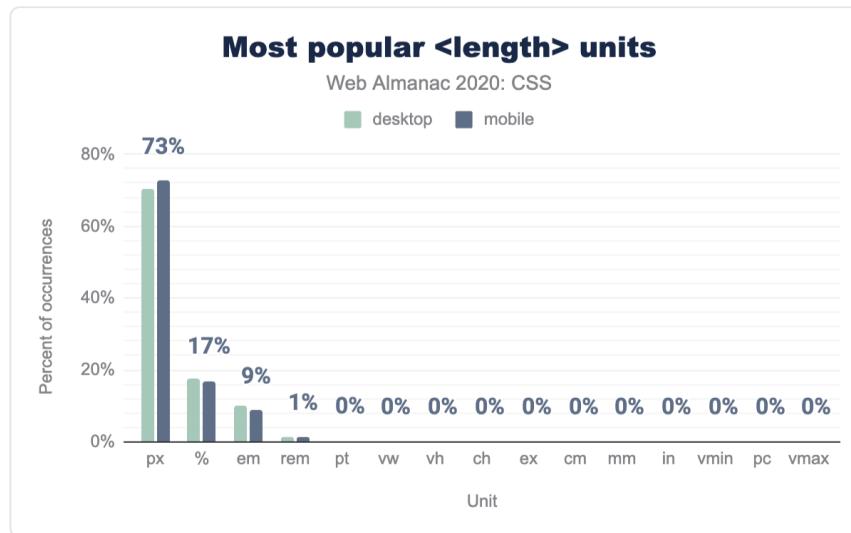


Figure 1.14. The most popular `<length>` units as a percent of occurrences.

How are these `px` distributed across properties? Is there any difference depending on the property? Most definitely. For example, as one might expect, `px` is far more popular in borders (80-90%) compared to font-related metrics such as `font-size`, `line-height` or `text-indent`. However, even for those, `px` usage vastly outnumbers any other unit. In fact, the only properties for which another unit (any other unit) is more used than `px` are `vertical-align` (55% `em`), `mask-position` (50% `em`), `padding-inline-start` (62% `em`), `margin-block-start` and `margin-block-end` (65% `em`), and the brand new `gap` with 62% `rem`.

One could easily argue that a lot of this content is just old, written before authors were more enlightened about using relative units to make their designs more adaptable and save themselves time down the line. However, this is easily debunked by looking at more recent

properties such as `grid-gap` (62% px).

Property	px	<number>	em	%	rem	pt
<code>font-size</code>	70%		2%	17%	6%	4%
<code>line-height</code>	54%		31%	13%	3%	
<code>border</code>	71%		27%	2%		
<code>border-radius</code>	65%		21%	3%	10%	
<code>text-indent</code>	32%		51%	8%	9%	
<code>vertical-align</code>	29%		12%	55%	4%	
<code>grid-gap</code>	63%		11%	9%	1%	16%
<code>mask-position</code>				50%	50%	
<code>padding-inline-start</code>	33%		5%	62%		
<code>gap</code>	21%		16%	1%		62%
<code>margin-block-end</code>	4%		31%	65%		
<code>margin-inline-start</code>	38%		46%	14%		1%

Figure 1.15. Unit usage by property.

Similarly, despite the much touted advantages of `rem` vs `em` for many use cases, and its universal browser support for years, the Web has still largely not caught up with it: the trusty `em` accounts for 87% of all font-relative units usage and `rem` trails far behind with 12%. We did see some usage of `ch` (width of the 'O' glyph) and `ex` (x-height of the font in use) in the wild, but very small (only 0.37% and 0.19% of all font-relative units).

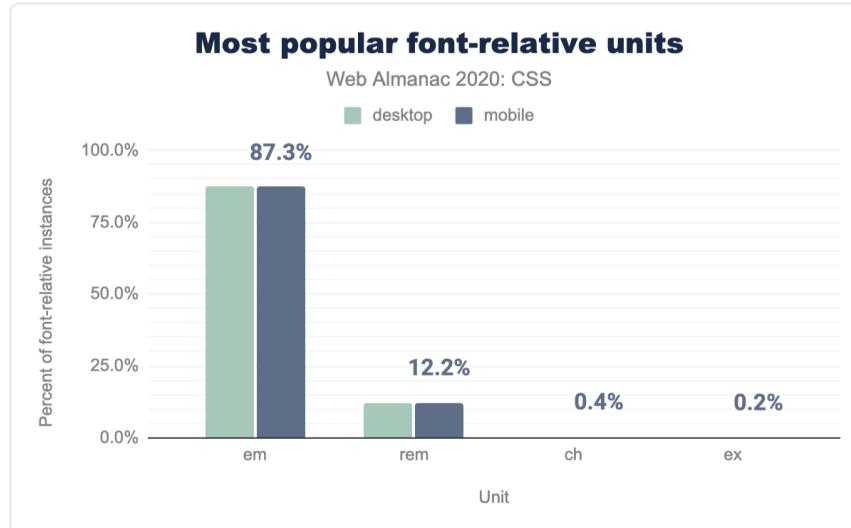


Figure 1.16. Relative share of font-relative units

Lengths are the only types of CSS values for which we can omit the unit when the value is zero, i.e. we can write `0` instead of `0px` or `0em` etc. Developers (or CSS minifiers?) are taking advantage of this extensively: Out of all `0` values, 89% were unitless.

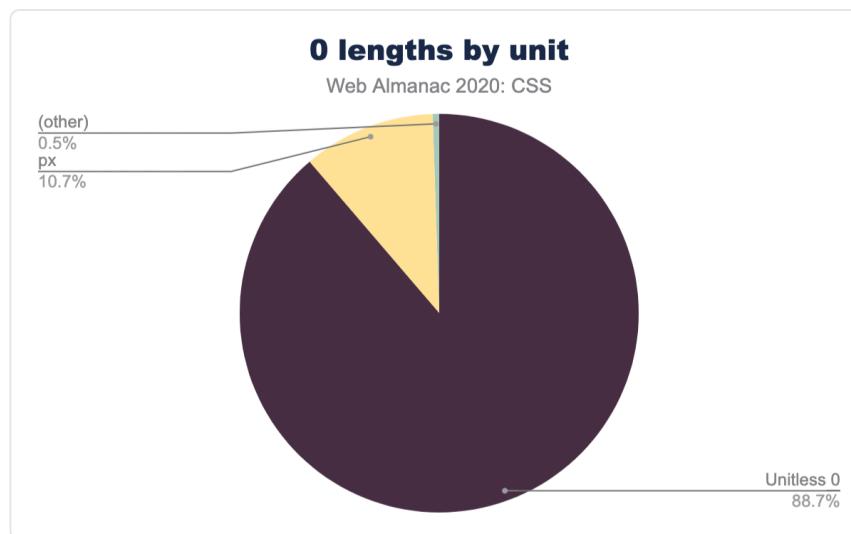


Figure 1.17. Relative popularity of 0 lengths by unit as a percent of occurrences on mobile pages.

Calculations

When the `calc()` function was introduced for performing calculations between different units in CSS, it was a revolution. Previously, only preprocessors were able to accommodate such calculations, but the results were limited to static values and unreliable, since they were missing the dynamic context that is often necessary.

Today, `calc()` has been supported by every browser for nine years already, so it comes as no surprise that it has been widely adopted with 60% of pages using it at least once. If anything, we expected even higher adoption than this.

`calc()` is primarily used for lengths, with 96% of its usage being concentrated in properties that accept `<length>` values, and 60% of that (58% of total usage) on the `width` property!

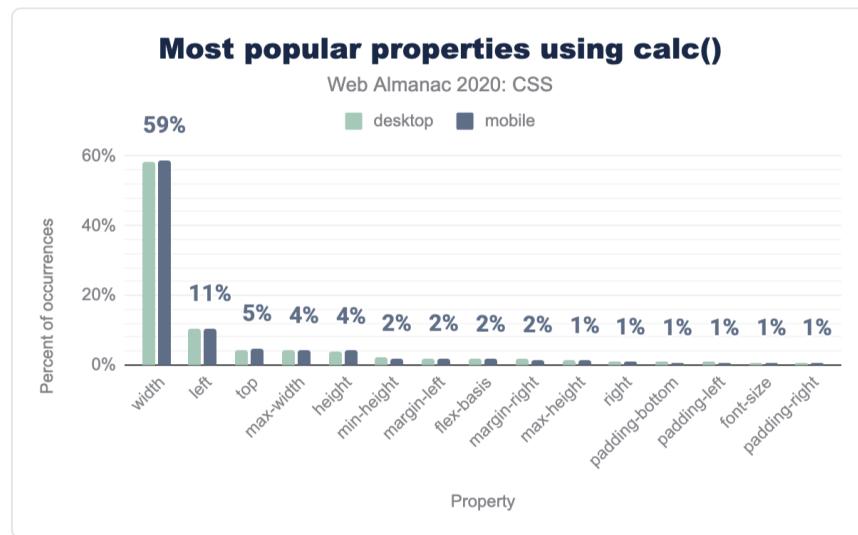


Figure 1.18. Relative popularity of properties that use `calc()` as a percent of occurrences.

It appears that most of this usage is to subtract pixels from percentages, as evidenced by the fact that the most common units in `calc()` are `px` (51% of `calc()` usage) and `%` (42% of `calc()` usage), and that 64% of `calc()` usage involves subtraction. Interestingly, the most popular length units with `calc()` are different than the most popular length units overall (e.g. `rem` is more popular than `em`, followed by viewport units), most likely due to the fact that code using `calc()` is newer.

Most popular units used in calc()

Web Almanac 2020: CSS

 desktop
 mobile


Figure 1.19. Relative popularity of units that use `calc()` as a percent of occurrences.

Most popular operators used in calc()

Web Almanac 2020: CSS

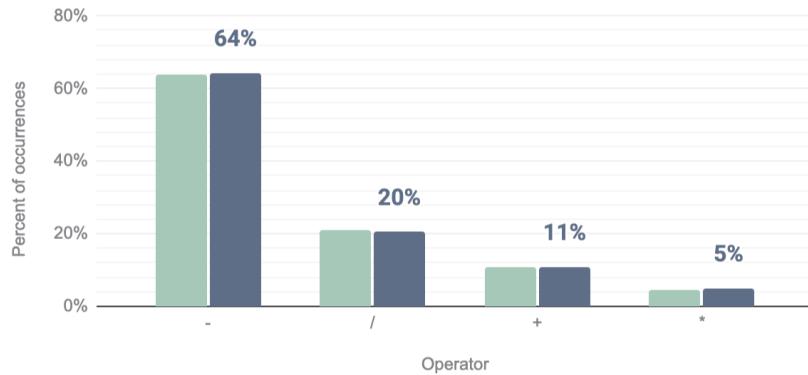
 desktop
 mobile


Figure 1.20. Relative popularity of operators that use `calc()` as a percent of occurrences.

Most calculations are very simple, with 99.5% of calculations involving up to 2 different units, 88.5% of calculations involving up to 2 operators and 99.4% of calculations involving one set of parentheses or fewer (3 out of 4 calculations include no parentheses at all).

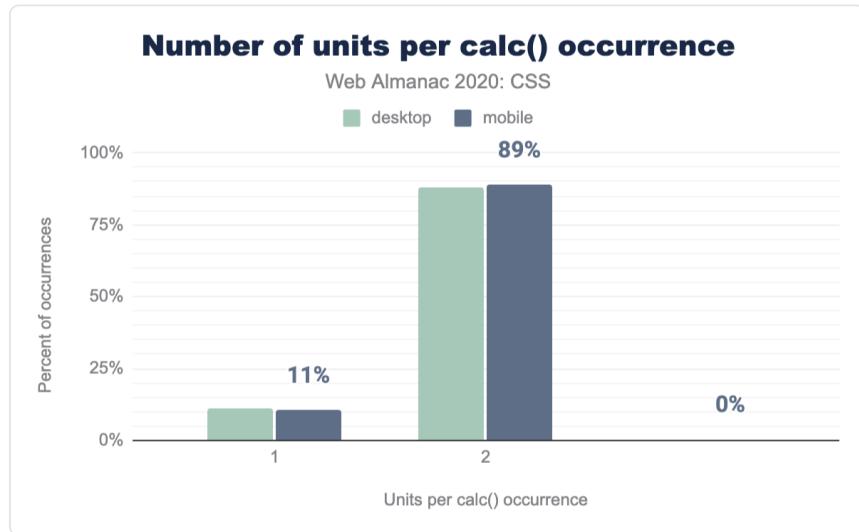


Figure 1.21. Distribution of the number of units per `calc()` occurrence.

Global keywords and `all`

For a long time, CSS only supported one global keyword: `inherit`, which enables the resetting of an inheritable property to its inherited value or reusing the parent's value for a given non-inheritable property. It turns out the former is far more common than the latter, with 81.37% of `inherit` usage being found on inheritable properties. The rest is mostly to inherit backgrounds, borders, or dimensions. The latter likely indicates layout struggles, as with the proper layout mode one rarely needs to force `width` and `height` to inherit.

The `inherit` keyword has been particularly useful for resetting the gory default link colors to the parent's text color, when we intend to use something other than color as an affordance for links. It is therefore no surprise that `color` is the most common property that `inherit` is used on. Nearly one third of all `inherit` usage is found on the `color` property. 75% of pages use `color: inherit` at least once.

While a property's *initial value* is a concept that has existed since CSS 1, it only got its own dedicated keyword, `initial`, to explicitly refer to it 17 years later, and it took another two years for said keyword to gain universal browser support in 2015. It is therefore no surprise that it is used far less than `inherit`. While the ol' `inherit` is found on 85% of pages, `initial` appears in 51% of them. Furthermore, there is a lot of confusion about what `initial` actually does, since `display` tops the list of properties most commonly used with `initial`, with `display: initial` appearing in 10% of pages. Presumably, the developers thought that this

resets `display` to its value from the user agent stylesheet value and were using it to toggle `display: none` on and off. However, the initial value of `display` is `inline`, so `display: initial` is just another way to write `display: inline` and has no context-dependent magical properties.

Instead, `display: revert` would have actually done what these developers likely expected and would have reset `display` to the UA value for the given element. However, `revert` is much newer: it was defined in 2015 and only gained universal browser support this year, which explains its underuse: it only appears in 0.14% of pages and half of its usage is `line-height: revert;`, found in recent versions of Wordpress' TwentyTwenty theme.

The last global keyword, `unset` is essentially a hybrid of `initial` and `inherit`. On inherited properties it becomes `inherit` and on the rest it becomes `initial`, essentially resetting the property across all cascade origins. Similarly, to `initial`, it was defined in 2013 and gained full browser support in 2015. Despite `unset`'s higher utility, it is used in only 43% of pages, whereas `initial` is used in 51% of pages. Furthermore, besides `max-width` and `min-width`, in every other property `initial` usage outweighs `unset` usage.

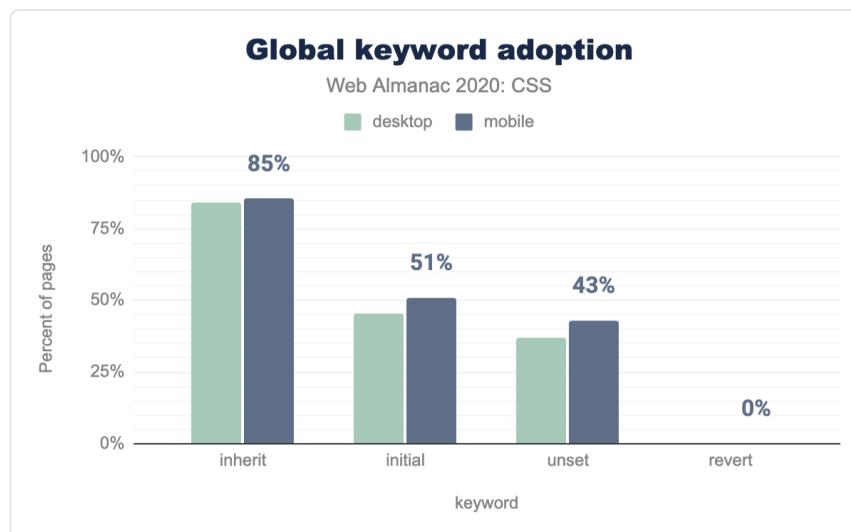


Figure 1.22. Adoption of global keywords as a percent of pages.

The `all` property was introduced in 2013 and gained near-universal support in 2016 (except Edge) and universal support earlier this year. It is a shorthand of nearly every property in CSS (except custom properties, `direction`, and `unicode-bidi`), and only accepts the four global keywords as values. It was envisioned as a one liner CSS reset, either as `all: unset` or `all: revert`, depending on what kind of reset we wanted. However, adoption is still very

low: we only found `all` on 477 pages (0.01% of all pages), and only used with the `revert` keyword.

Color

They say the old jokes are the best, and that goes for colors too. The original, cryptic, `#rrggbb` hex syntax remains the most popular way to specify a color in CSS in 2020: Half of all colors are written that way. The next most popular format is the somewhat shorter `#rgb` three-digit hex format at 26%. While it is shorter, it is also able to express way fewer colors; only 4096, out of the 16.7 million sRGB values.

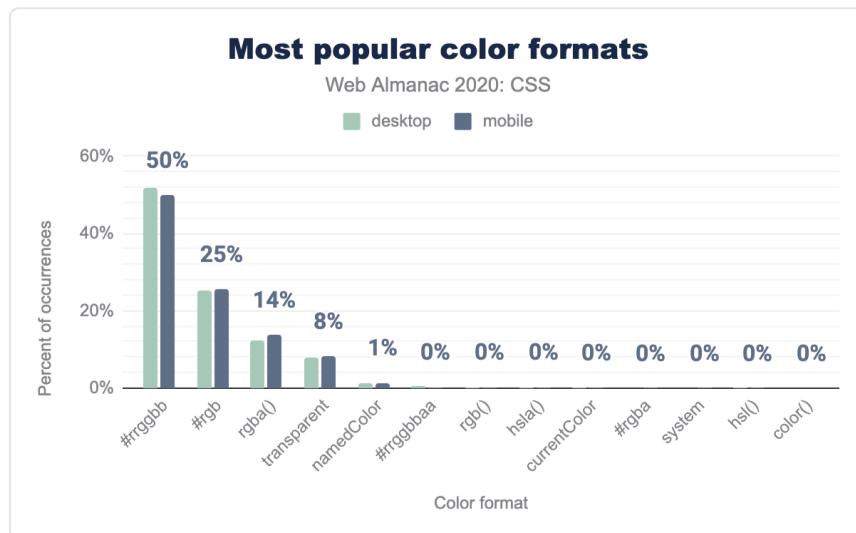


Figure 1.23. Relative popularity of color formats as a percent of occurrences.

Similarly, 99.89% of functionally specified sRGB colors are using the since-forever legacy format with commas `rgb(127, 255, 84)` rather than the new comma-less form `rgb(127 255 84)`. Because, despite all modern browsers accepting the new syntax, changing offers zero advantage to developers.

So why do people stray from these tried and true formats? To express alpha transparency. This is clear when you look at `rgba()`, which is used 40 times more than `rgb()` (13.82% vs 0.34% of all colors) and `hsla()`, which is used 30 times more than `hsl()` (0.25% vs 0.01% of all colors).

And yes, these numbers also show that despite the much-vaunted (but largely incorrect) easily-

understood or easily-modified advantages of HSL, in practice it is used far less than RGB.

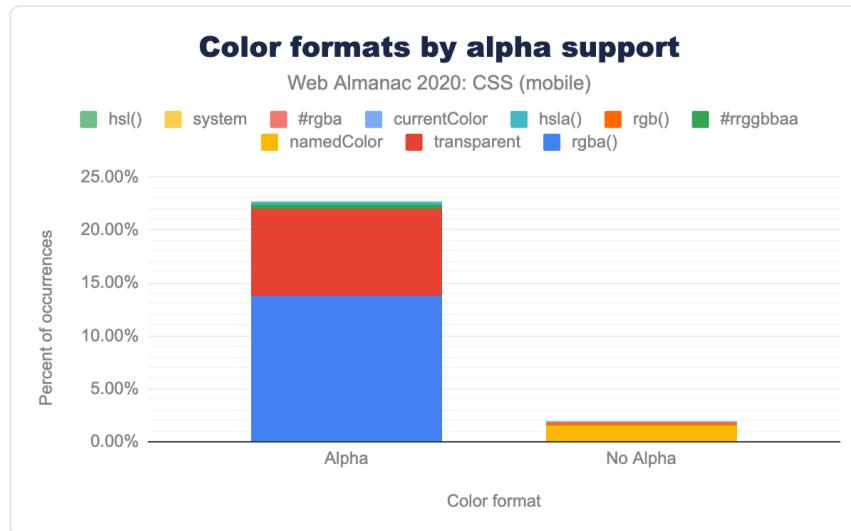


Figure 1.24. Relative popularity of color formats grouped by alpha support as a percent of occurrences on mobile pages (excluding `#rrggbb` and `#rgb`).

What about named colors? The keyword `transparent`, which is just another way to say `rgb(0 0 0 / 0)`, is most popular, at 8.25% of all sRGB values (66% of all named-color usage); followed by all the named (X11) colors – I’m looking at you, `papayawhip` – at 1.48%. The most popular of these were the easily understood names like `white`, `black`, `red`, `gray`, `blue`. `Whitesmoke` was the most common of the non-ordinary names (sure, we can visualize whitesmoke, right) while the likes of `gainsboro`, `lightCoral` and `burlywood` were used way less. We can understand why, you need to look them up to see what they actually mean.

And if you are going for fanciful color names, why not define your own with CSS Custom properties? `--intensePurple` and `--corporateBlue` mean whatever you need them to mean. This probably explains why 50% of Custom Properties are used for colors.

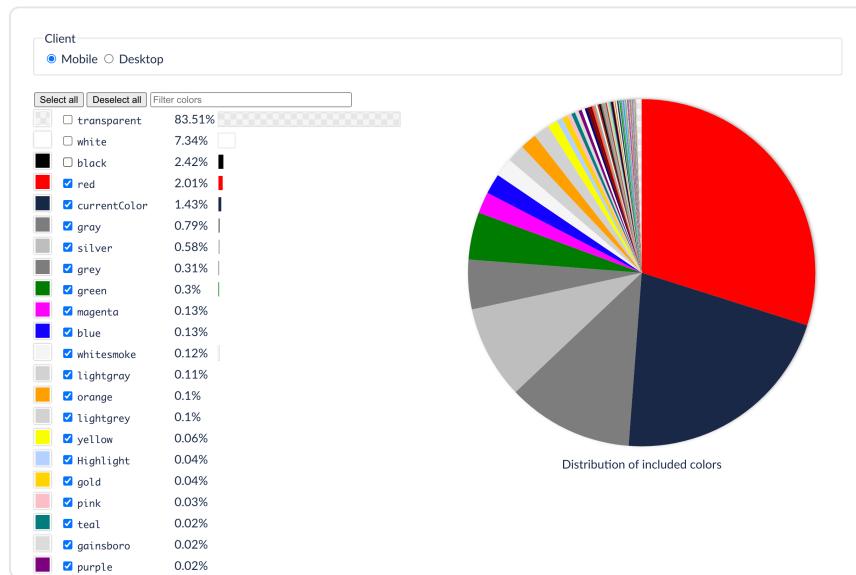


Figure 1.25. Interactively explore the color keyword usage data with this app!

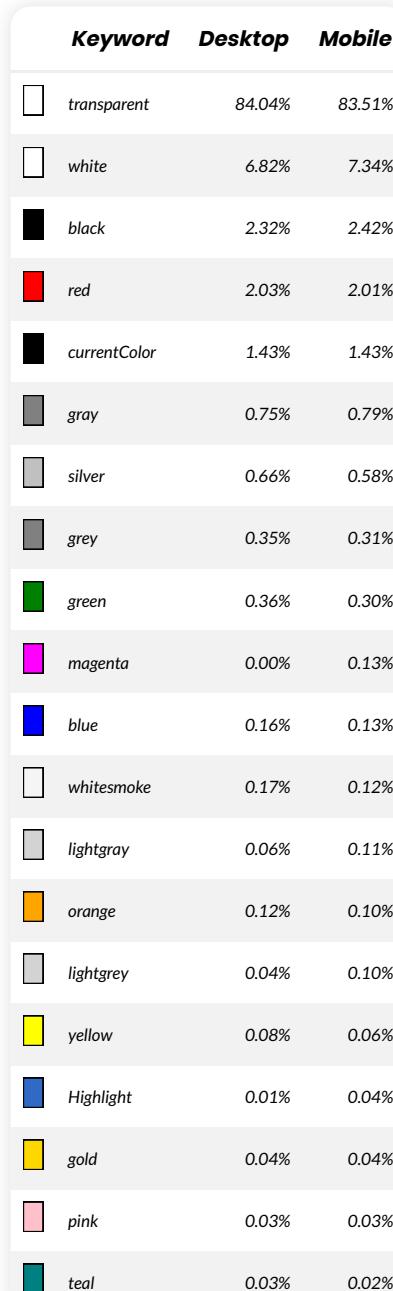


Figure 1.26. Relative popularity of color keywords as a percent of occurrences.

And, lastly, the once-deprecated, now partially un-deprecated system colors like `Canvas` and `ThreeDDarkShadow`: These were a terrible idea, introduced to emulate the typical user interface of things like Java or Windows 95, and already unable to keep up with Windows 98, they soon fell by the wayside. Some sites use these system colors to try and fingerprint you, a loophole that we are trying to close as we speak. There are *few good reasons* to use them, and most websites (99.99%) don't, so we are all good.

The rather useful value `currentColor`, surprisingly, trailed at 0.14% of all sRGB colors (1.62% of all named colors).

All the colors we discussed so far have one thing in common: sRGB, the standard color space for the Web (and for High Definition TV, which is where it came from). Why is that so bad? Because it can only display a limited range of colors: your phone, your TV, and probably your laptop are able to display much more vivid colors due to advances in display technology. Displays with wide color gamut, which used to be reserved for well-paid professional photographers and graphic designers, are now available to everyone. Native apps use this capability, as do digital movies and streaming TV services, but until recently the Web was missing out.

And we are still missing out. Despite being implemented in Safari in 2016, the use of `display-p3` color in Web pages is vanishingly small. Our crawl of the Web found only 29 mobile and 36 desktop pages (!) using it. (And more than half of those were syntax errors, mistakes, or attempts to use the never-implemented `color-mod()` function). We were curious why.

Compatibility, right? You don't want things to break? No. In the stylesheets we examined, we found solid use of fallback: with document order, the cascade, `@supports`, the `color-gamut` media query, all that good stuff. So in a style sheet we would see the color the designer wanted, expressed in `display-p3`, and also a fallback sRGB color. We computed the visible difference (a calculation called $\Delta E 2000$) between the desired and fallback color and this was typically quite modest. A small tweak. A careful exploration. In fact, 37.6% of the time, the color specified in `display-p3` actually fell inside the range of colors (the gamut) that sRGB can manage.

sRGB	display-p3	ΔE2000	In gamut
<code>rgba(255,205,63,1)</code>	 <code>color(display-p3 1 0.80 0.25 / 1)</code>	3.880	false
<code>rgba(120,0,255,1)</code>	 <code>color(display-p3 0.47 0 1 / 1)</code>	1.933	false
<code>rgba(121,127,132,1)</code>	 <code>color(display-p3 0.48 0.50 0.52 / 1)</code>	0.391	true
<code>rgba(200,200,200,1)</code>	 <code>color(display-p3 0.78 0.78 0.78 / 1)</code>	0.274	true
<code>rgba(97,97,99,1)</code>	 <code>color(display-p3 0.39 0.39 0.39 / 1)</code>	1.474	true
<code>rgba(0,0,0,1)</code>	 <code>color(display-p3 0 0 0 / 1)</code>	0.000	true
<code>rgba(255,255,255,1)</code>	 <code>color(display-p3 1 1 1 / 1)</code>	0.015	false
<code>rgba(84,64,135,1)</code>	 <code>color(display-p3 0.33 0.25 0.53 / 1)</code>	1.326	true
<code>rgba(131,103,201,1)</code>	 <code>color(display-p3 0.51 0.40 0.78 / 1)</code>	1.348	true
<code>rgba(68,185,208,1)</code>	 <code>color(display-p3 0.27 0.75 0.82 / 1)</code>	5.591	false
<code>rgb(255,0,72)</code>	 <code>color(display-p3 1 0 0.2823 / 1)</code>	3.529	false
<code>rgba(255,205,63,1)</code>	 <code>color(display-p3 1 0.80 0.25 / 1)</code>	3.880	false
<code>rgba(241,174,50,1)</code>	 <code>color(display-p3 0.95 0.68 0.17 / 1)</code>	4.701	false
<code>rgba(245,181,40,1)</code>	 <code>color(display-p3 0.96 0.71 0.16 / 1)</code>	4.218	false
<code>rgb(147,83,255)</code>	 <code>color(display-p3 0.58 0.33 1 / 1)</code>	2.143	false
<code>rgba(75,3,161,1)</code>	 <code>color(display-p3 0.29 0.01 0.63 / 1)</code>	1.321	false
<code>rgba(255,0,0,0.85)</code>	 <code>color(display-p3 1 0 0 / 0.85)</code>	7.115	false
<code>rgba(84,64,135,1)</code>	 <code>color(display-p3 0.33 0.25 0.53 / 1)</code>	1.326	true
<code>rgba(131,103,201,1)</code>	 <code>color(display-p3 0.51 0.40 0.78 / 1)</code>	1.348	true
<code>rgba(68,185,208,1)</code>	 <code>color(display-p3 0.27 0.75 0.82 / 1)</code>	5.591	false
<code>#6d3bff</code>	 <code>color(display-p3 0.427 .231 1)</code>	1.584	false
<code>#03d658</code>	 <code>color(display-p3 0.012 .839 .345)</code>	4.958	false
<code>#ff3900</code>	 <code>color(display-p3 1 .224 0)</code>	7.140	false
<code>#7cf8b3</code>	 <code>color(display-p3 0.486 .973 .702)</code>	4.284	true
<code>#f8f8f8</code>	 <code>color(display-p3 0.973 .973 .973)</code>	0.028	true
<code>#e3f5fd</code>	 <code>color(display-p3 0.875 .945 .976)</code>	1.918	true
<code>#e74832</code>	 <code>color(display-p3 0.905882353 .282352941 .196078431 / 1)</code>	3.681	true

Figure 1.27. This table shows the fallback sRGB colors (plus a color swatch), then the display-p3 colors. A color difference ($\Delta E2000$) of 1 is barely visible, while 5 is clearly distinct.

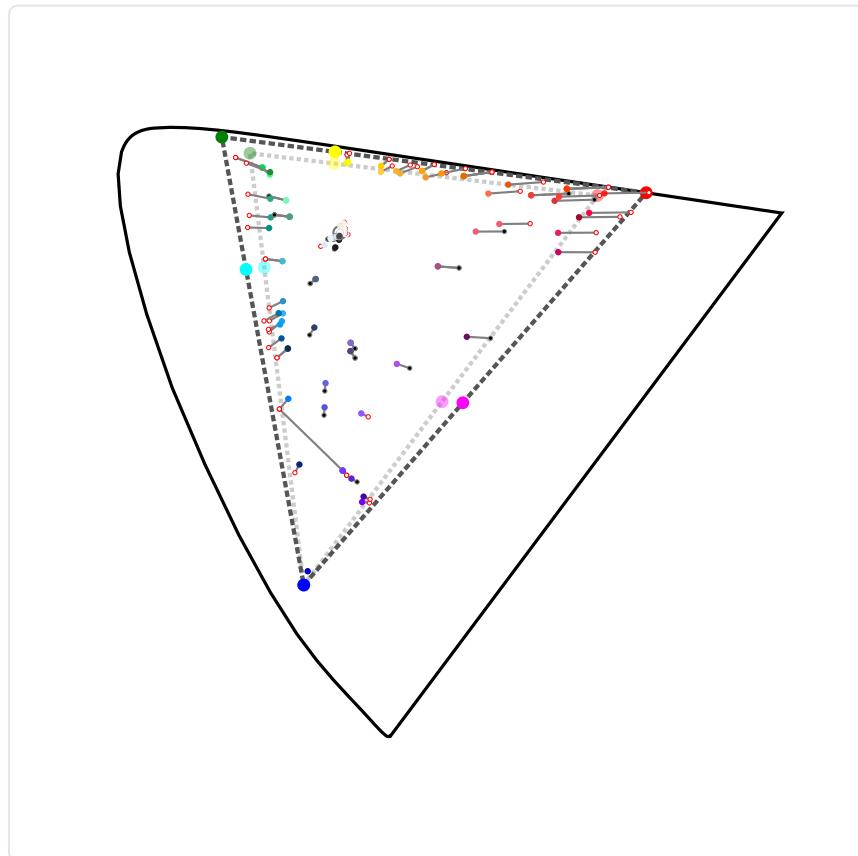


Figure 1.28. uv chromaticity of specified display-p3 colors and their fallbacks.

The purplish colors are similar in sRGB and display-p3, perhaps because both those color spaces have the same blue primary. Various reds, orange-yellows, and greens are near the sRGB gamut boundary (nearly as saturated as possible) and map to analogous points near the display-p3 gamut boundary.

There seem to be two reasons why the Web is still trapped in sRGB land. The first is lack of tools, lack of good color pickers, lack of understanding of what more vivid colors are available. But the major reason, we think, is that to date Safari is the only browser to implement it. This is changing, rapidly - Chrome and Firefox are both implementing right now - but until that support ships, probably using display-p3 is too much effort for too little gain because only 17% of viewers will see those colors. Most people will see the fallback. So current usage is a subtle shift in color vibrancy, rather than a big difference.

It will be interesting to see how the use of `display-p3` color (other options exist, but this is the only one we found in the wild) changes over the next year or two.

Because *wide color gamut* (WCG) is only the beginning. The TV and movie industry has already moved past P3 to an even wider gamut, rec2020; and also a wider range of lightness, from blinding reflections to deepest shadows. *High Dynamic Range* (HDR) has already arrived in the home, especially on games, streaming TV and movies. The Web has a bunch of catching up to do.

Gradients

Despite minimalism and flat design being all the rage, CSS gradients are used in 75% of pages. As expected, nearly all gradients are used in backgrounds. 74.45% of pages specify gradients in backgrounds, but only 7% in **any** other property.

Linear gradients are 5 times more popular than radial ones, appearing in almost 73% of pages, compared to 15% for radial gradients. The difference in popularity is so staggering, that even `-ms-linear-gradient()`, which **was never needed** (IE10 supported gradients both with and without the `-ms-` prefix), is more popular than `radial-gradient()`! The newly supported `conic-gradient()` is even more underutilized, appearing in only 652 desktop pages (0.01%) and 848 mobile pages (0.01%), which is expected, since Firefox has only just shipped its implementation to the stable channel.

Repeating gradients of all types are fairly underused too, with `repeating-linear-gradient()` appearing in only 3% of pages and the others trailing behind even more (`repeating-conic-gradient()` is only used in 21 pages!).

Prefixed gradients are also still very common, even though prefixes haven't been needed in gradients since 2013. It is notable that `-webkit-gradient()` is still used in half of all websites, even though it hasn't been needed since 2011. And `-webkit-linear-gradient()` is still the second most used gradient function of all, appearing in 57% of websites, with the other prefixed forms also being used in a third to half of pages.

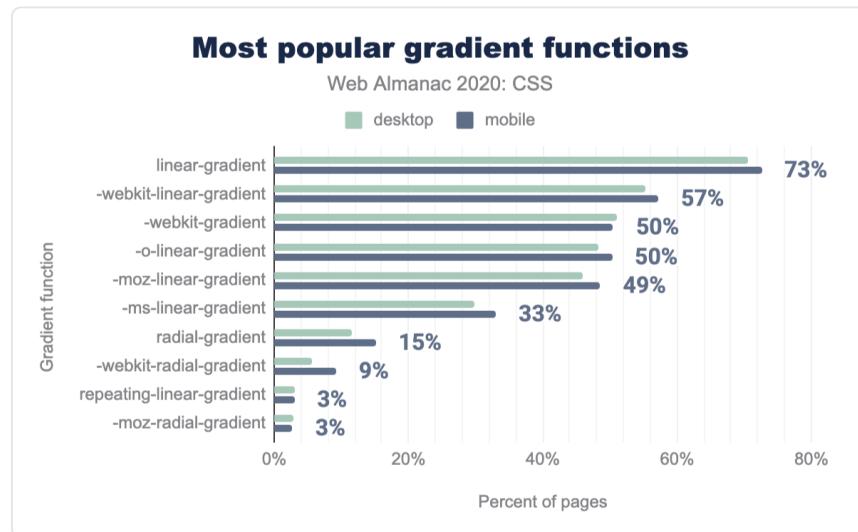


Figure 1.29. The most popular gradient functions as a percent of pages.

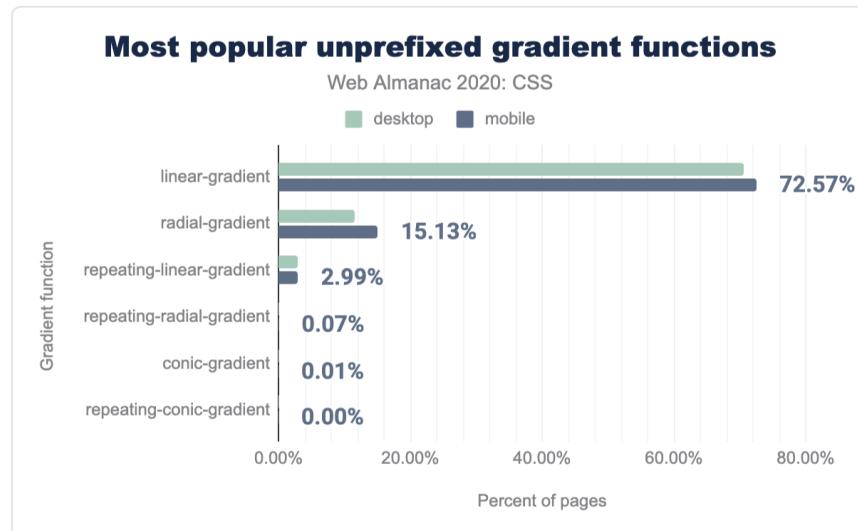


Figure 1.30. The most popular gradient functions as a percent of pages, omitting vendor prefixes.

Using color stops with different colors in the same position (hard stops) to create stripes and other patterns is a technique first popularized in 2010 (by Lea Verou), which by now has many interesting variations, including some really cool ones with blend modes. While it may seem like a hack, hard stops are found in 50% of pages, indicating a strong developer need for lightweight graphics from within CSS without resorting to image editors or external SVG.

Interpolation hints (or as Adobe, who popularized the technique, calls them: “midpoints”) are found on 22% of pages, despite near universal browser support since 2015. Which is a shame, because without them, the color stops are connected by straight-lines in the colorspace, rather than smooth curves. This low usage probably reflects a misunderstanding of what they do, or how to use them; contrast this with CSS transitions and animations, where easing functions (which do much the same thing, i.e. connect the keyframes with curves rather than jerky straight lines) are much more commonly used (80% of transitions). “Midpoints” is not a very understandable description, and “interpolation hint” sounds like you are helping the browser to do simple arithmetic.

Most gradient usage is rather simple, with over 75% of gradients found across the entire dataset only using 2 color stops. In fact, fewer than half of pages contain even a single gradient with more than 3 color stops!

The gradient with the most color stops is this one with 646 stops! So pretty! This is almost certainly generated, and the resulting CSS code is 8KB, so a 1px tall PNG would likely have done the job as well, with a smaller footprint.



Figure 1.31. The gradient with the most color stops, 646.

Layout

Flexbox and Grid adoption

In the 2019 edition, 41% of pages across mobile and desktop were reported as containing Flexbox properties. In 2020, this number has grown to 63% for mobile and 65% for desktop. With the number of legacy sites developed before Flexbox was a viable tool still in existence, we can safely say there is wide adoption of this layout method.

If we look at Grid layout, the percentage of sites using Grid layout has grown to 4% for mobile and 5% for desktop. Usage has doubled since last year, but still lags behind flex layout.

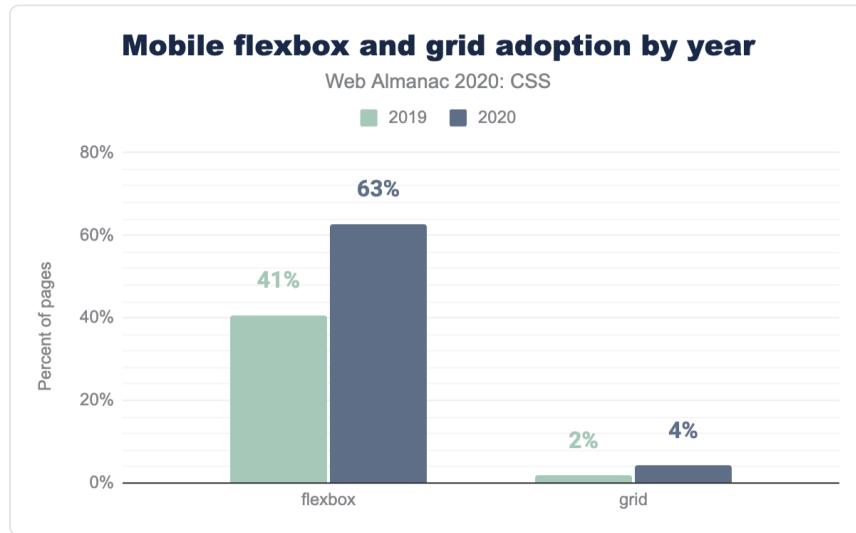


Figure 1.32. Adoption of Flexbox and grid by year as a percent of mobile pages.

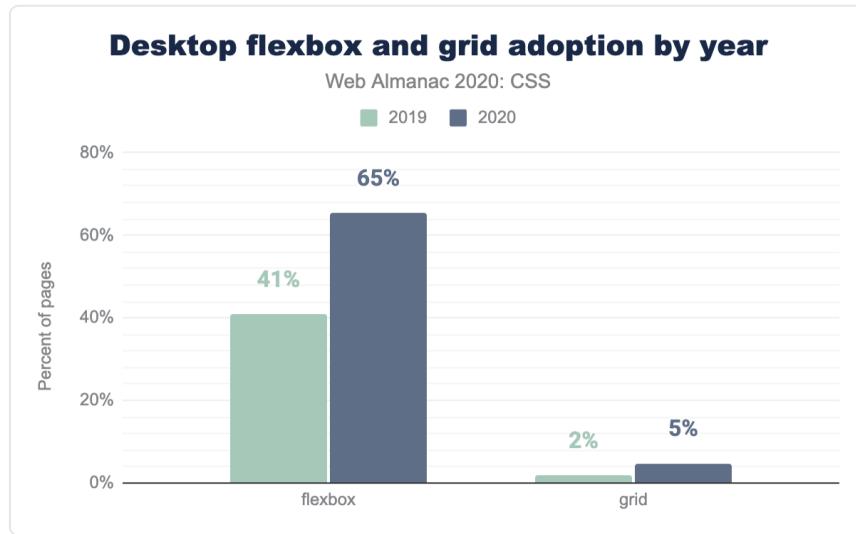


Figure 1.33. Adoption of flexbox and grid by year as a percent of desktop pages.

Note that unlike most other metrics in this chapter this is actual measured Grid usage, and not just grid-related properties and values that are specified in a stylesheet and potentially not used. While at first glance this may seem more accurate, one thing to keep in mind is that HTTP Archive crawls homepages, so this data may be skewed lower due to grids often appearing more in internal pages. So, let's look at another metric as well: how many pages specify

`display: grid` and `display: flex` in their stylesheets? That metric puts Grid layout at significantly higher adoption, with 30% of pages using `display: grid` at least once. It does not however affect the number for Flexbox as significantly, with 68% of pages specifying `display: flex`. While this sounds like impressively high adoption for Flexbox, it's worth noting that CSS tables are still far more popular with 80% of pages using table display modes! Some of this usage may be due to certain types of clearfix which use `display: table`, and not for actual layout.

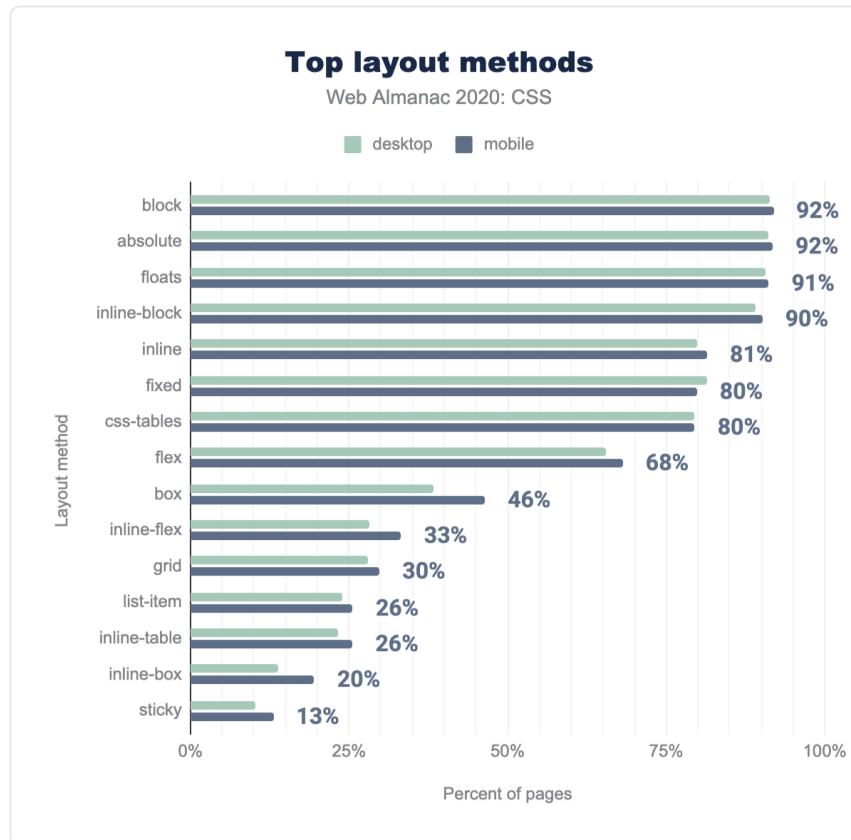


Figure 1.34. Layout modes and percentage of pages they appear on. This data is a combination of certain values from the `display`, `position`, and `float` properties.

Given that Flexbox was usable in browsers earlier than Grid layout, it is likely that some of the Flexbox usage is for setting up a grid system. In order to use Flexbox as a grid, authors need to disable some of the inherent flexibility of Flexbox. To do this you set the `flex-grow` property to `0`, then size flex items using percentages. Using this information we were able to report that 19% of sites both on desktop and mobile were using Flexbox in this grid-like way.

The reasons for choosing Flexbox over grid are frequently cited as browser support, given that Grid layout was not supported in Internet Explorer. In addition, some authors may well not have learned Grid layout yet or are using a framework with a Flexbox-based grid system. The Bootstrap framework currently uses a Flexbox-based grid, in common with several other popular framework choices.

Usage of different Grid layout techniques

The Grid layout specification gives a number of ways to describe and define layout in CSS. The most basic usage involves laying items out from one grid line to another. What about naming lines, or use of grid-template-areas?

For named lines, we checked for the presence of square brackets in a track listing. The name or names being placed inside square brackets.

```
.wrapper {  
  display: grid;  
  grid-template-columns: [main-start] 1fr [content-start] 1fr  
  [content-end] 1fr [main-end];  
}
```

The result of this showed that 0.23% of grid-using pages on mobile had named lines, and 0.27% on desktop.

The grid-template-areas feature, allowing authors to name grid items then place them on the grid as the value of the grid-template-areas property, fared a little better. Of grid-using sites, 19% on mobile and 20% on desktop were using this method.

These results show that not only is Grid layout usage still relatively low on production websites, the usage of it is relatively straightforward. Authors are choosing to use the simple line-based placement over methods which would allow them to name lines and areas. While there is nothing wrong in choosing to do so, I wonder if slow adoption of Grid layout is partly due to the fact that authors haven't yet realized the power of these features. If Grid layout is seen as essentially Flexbox with poor browser support, this would certainly make it a less compelling choice.

Multiple-column layout

The multiple-column layout specification enables laying out of content in columns, much as in a

newspaper. While popular in CSS as used for print, it is less useful on the web due to the risk of creating a situation where a reader needs to scroll up and down to read the content. Based on the data, however, there are significantly more pages using multicol than Grid layout with 15.33% on the desktop and 14.95% on mobile. While basic multicol properties are well supported, more complex usage and controlling column breaks with fragmentation has patchy support. Considering this, it was quite surprising to see how much usage there is.

Box sizing

It is useful to know how big the boxes on your page are going to be, but with the standard CSS box model adding padding and border onto the size of the content-box, the size you gave your box is smaller than the box rendered on your page. While we can't change history, the box-sizing property allows authors to switch to applying the specified size to the border-box, so the size you set is the size you see rendered. How many sites are using the box-sizing property? Most of them! The box-sizing property appears in 83.79% of desktop CSS and 86.39% on mobile.

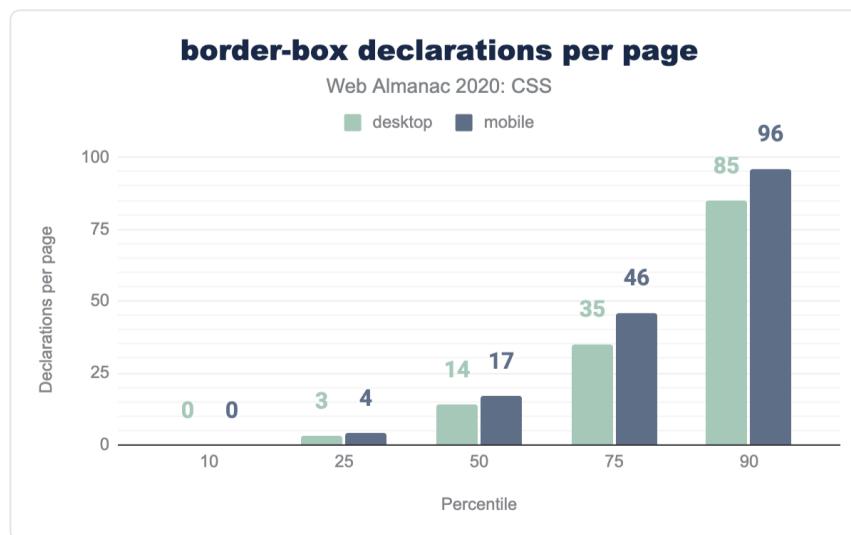


Figure 1.35. Distribution of the number of `border-box` declarations per page.

The median desktop page has 14 box-sizing declarations. Mobile has 17. Perhaps due to component systems inserting the declaration per component, rather than globally as a rule for all elements in the stylesheet.

Transitions and animations

Transitions and animations have overall become very popular with the `transition` property being used on 81% of all pages and `animation` on 73% of mobile pages and 70% of desktop pages. It is somewhat surprising that usage is not lower on mobile, where one would expect that conserving battery power would be a priority. On the other hand, CSS animations are *far* more battery efficient than JS animation, especially the majority of them that just animate transforms and opacity (see next section).

The single most common transition property specified is `all`, used in 41% of pages. This is a little baffling because `all` is the initial value, so it does not actually need to be explicitly specified. After that, fade in/out transitions appear to be the most common type, used in over one third of crawled pages, followed by transitions on the `transform` property (most likely spin, scale, movement transitions). Surprisingly, transitioning `height` is much more popular than transitioning `max-height`, even though the latter is a commonly taught workaround when the start or end height is unknown (auto). It was also surprising to see significant usage for the `scale` property (2%), despite its lack of support beyond Firefox. Intentional usage of cutting edge CSS, a typo, or a misunderstanding of how to animate transforms?

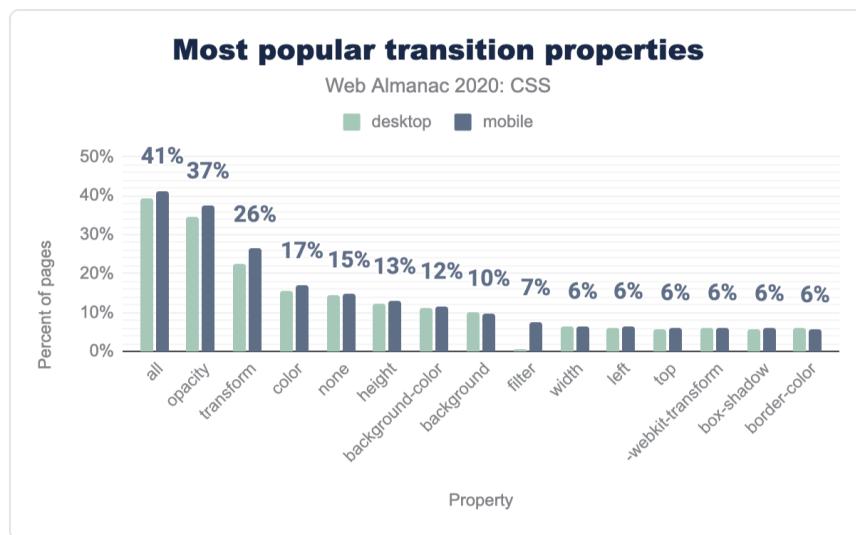


Figure 1.36. Adoption of transition properties as a percent of pages.

We were glad to discover that most of these transitions are fairly short, with the median transition duration being only 300ms, and 90% of websites having median durations of less than half a second. This is generally good practice, as longer transitions can make a UI feel sluggish, while a short transition communicates a change without getting in the way.

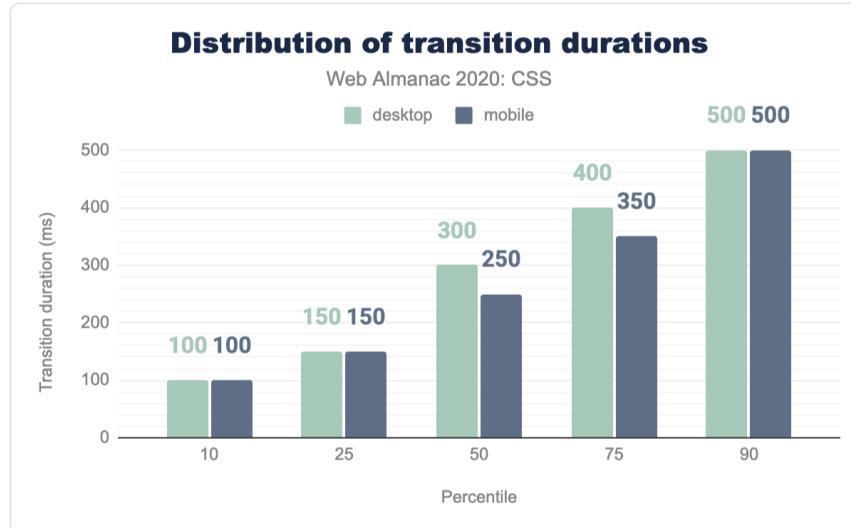


Figure 1.37. Distribution of transition durations.

The specification authors got it right! `Ease` is the most popular timing function specified, even though it's the default so it can actually be omitted. Perhaps people explicitly specify the defaults because they prefer the self documenting verbosity, or — perhaps more likely — because they don't know that they are defaults. Despite the drawbacks of linearly progressing animation (it tends to look dull and unnatural), `linear` is the second most highly used timing function with 19.1%. It is also interesting that the built-in easing functions accommodate over 87% of all transitions: only 12.7% chose to specify a custom easing via `cubic-bezier()`.

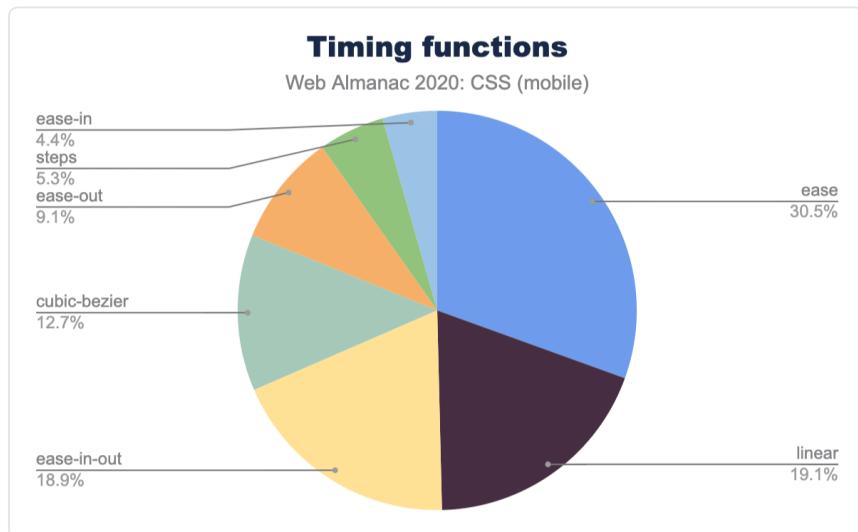


Figure 1.38. Relative popularity of timing functions as a percent of occurrences on mobile pages.

A major driver of animation adoption seems to be Font Awesome, as evidenced by the animation name `fa-spin` appearing in 1 out of 4 pages and thus topping the list of most popular animation names. While there are a wide variety of animation names, it appears that most of them fall into only a few basic categories, with 1 in 5 animations being some kind of spin. That may also explain the high percentage of linearly progressing transitions & animations: if we want a smooth perpetual rotation, `linear` is the way to go.

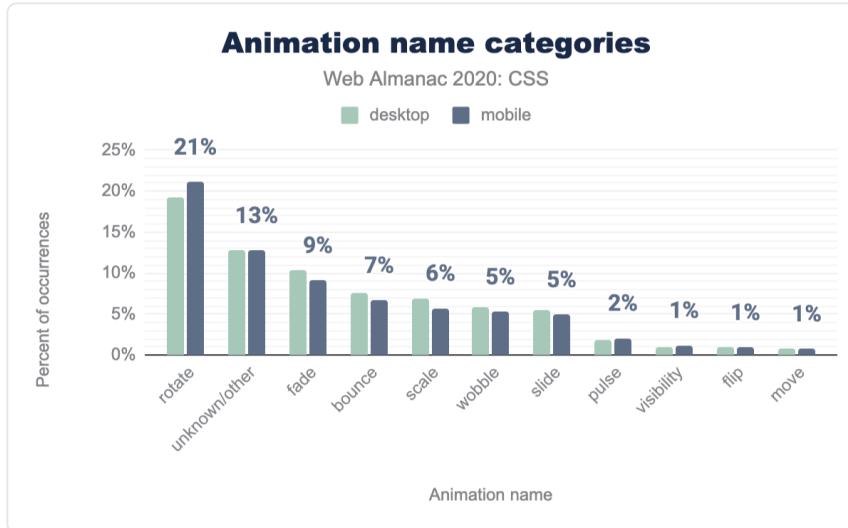


Figure 1.39. Relative popularity of the categories of animation names used as a percent of occurrences.

Visual effects

Blend modes

Last year, 8% of pages were using blend modes. This year, adoption has increased significantly, with 13% of pages using blend modes on elements (`mix-blend-mode`), and 2% in backgrounds (`background-blend-mode`).

Filters

Adoption of filters has remained high, with the `filter` property making an appearance in 79.43% of pages. While at first this was quite exciting, a lot of it is likely to be old IE DX filters, which shared the same property name. When we only took into account valid CSS filters that Blink recognizes, usage drops to 22% for mobile and 20% for desktop, with `blur()` being the most popular filter type, appearing in 4% of pages.

Another filter property, `backdrop-filter`, allows us to apply filters to only the area *behind* an element, which is incredibly useful for improving contrast on translucent backgrounds, and creating the elegant "frosted glass" effect we've come to know from many native UIs. While not nearly as popular as `filter`, we found `backdrop-filter` in 6% of pages.

The `filter()` function allows us to apply a filter only on a particular image, which can be extremely useful for backgrounds. Sadly, it is currently only supported by Safari. We did not find any usage of `filter()`.

Masks

A decade ago, we got masks in Safari with `-webkit-mask-image` and it was exciting. Everyone and their dog were using them. We eventually got a spec and a set of unprefixed properties closely modeled after the WebKit prototype, and it seemed a matter of time until masking became standard, with a consistent syntax across all browsers. Fast forward 10 years later, and the unprefixed syntax is still not supported in Chrome or Safari, meaning its available on less than 5% of users' browsers worldwide. It is therefore no surprise that `-webkit-mask-image` is still more popular than its standard counterpart, being found in 22% of pages. However, despite its very poor support, `mask-image` is found on 19% of pages. We see a similar pattern across most other masking properties with the unprefixed versions appearing in almost as many pages as the `-webkit-` ones. Overall, despite them falling out of hype, masks are still found in nearly a quarter of the Web, indicating that the use cases are still there, despite lack of implementer interest (hint, hint!).

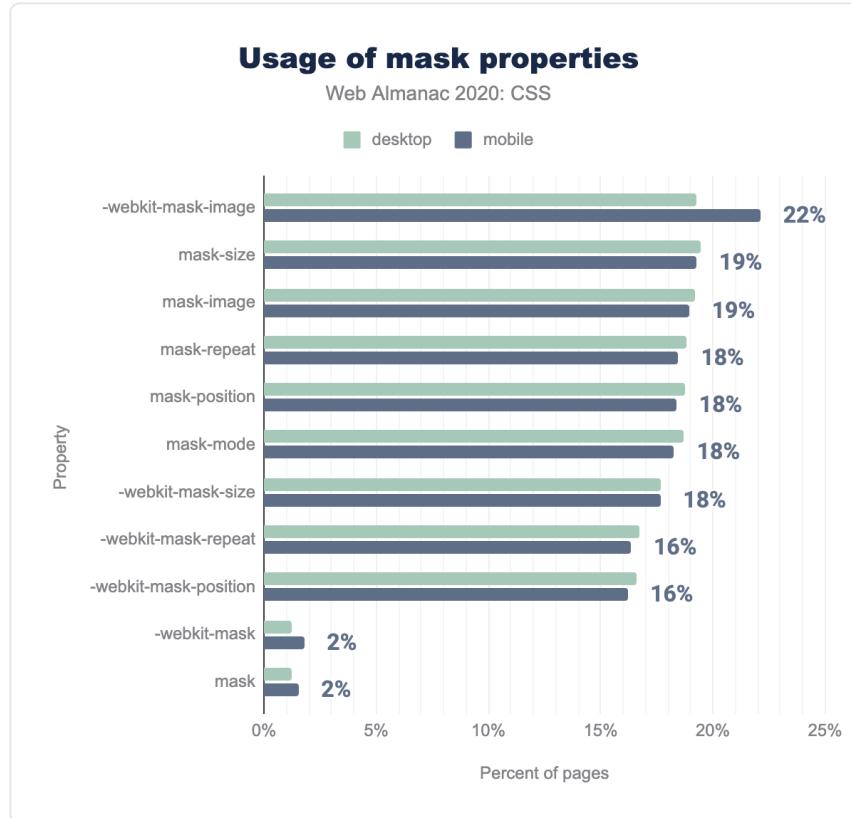


Figure 1.40. Relative popularity of animation name categories as a percent of occurrences.

Clipping paths

Around the same time masks got popular, another similar but simpler property (originally from SVG) started making the rounds: `clip-path`. However, unlike masks, it had a brighter fate. It got standardized fairly quickly, and got support across the board relatively fast, with the last holdout being Safari which dropped the prefix in 2016. Today, it is found on 19% of pages unprefixed and 13% with the `-webkit-` prefix.

Responsive design

Making sites that cope with the many different screen sizes and devices that browse the web has become somewhat easier with the built-in flexible and responsive new layout methods such

as Flexbox and Grid. These layout methods are usually further enhanced with the use of media queries. The data shows that 80% of desktop sites and 83% of mobile sites use media queries that are associated with responsive design, such as `min-width`.

Which media features are people using?

As you might expect, the most common media features in use are the viewport size features which have been in use since the early days of responsive web design. The percentage of sites checking for `max-width` is 78% for both desktop and mobile. A check for `min-width` features on 75% of mobile and 73% of desktop sites.

The `orientation` media feature, which allows authors to differentiate their layout based on whether the screen is portrait or landscape, can be found on 33% of all sites.

We are seeing some newer media features come up in the statistics. The `prefers-reduced-motion` media feature provides a way to check if the user has requested reduced motion, so that websites can adjust the amount of animation they use. This can be turned on either explicitly, through a user-controlled operating system setting, or implicitly, for example due to decreasing battery level. 24% of sites are checking for this feature.

In other good news, newer features from the Media Queries Level 4 specification are starting to appear. On mobile 5% of sites are checking for the type of pointer the user has. A `coarse` pointer indicates they are using a touchscreen, whereas a `fine` pointer indicates a pointing device. Understanding the way a user is interacting with your site is often just as helpful, if not more helpful, than looking at screen size. A person might be using a small screen device with a keyboard and mouse, or a high resolution large screen device with a touchscreen and benefit from larger hit areas.

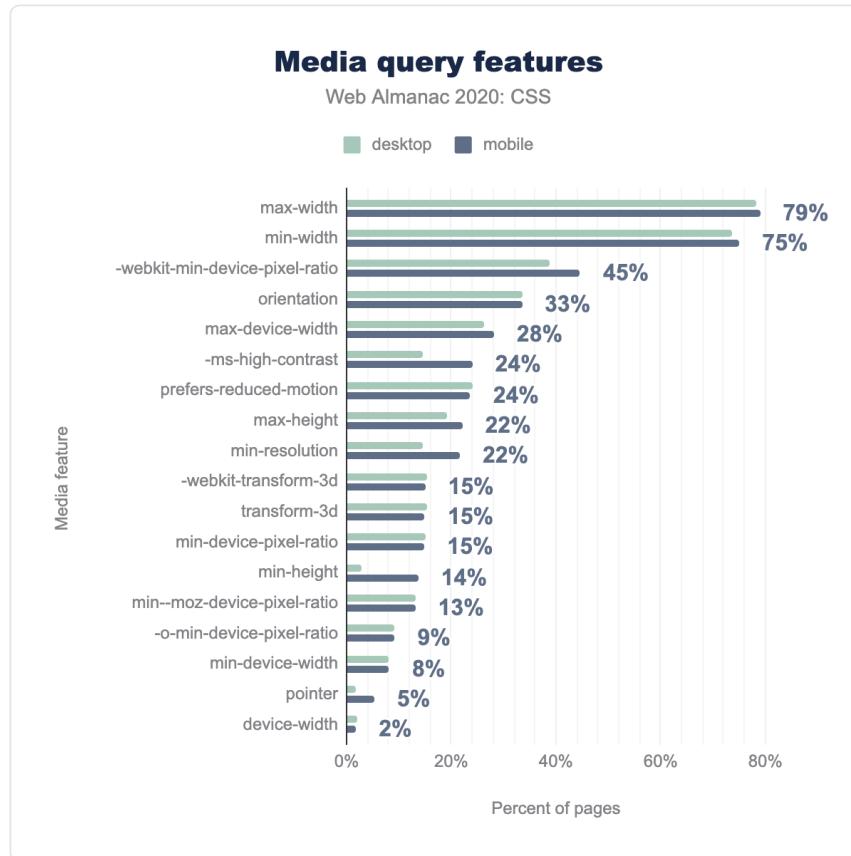


Figure 1.41. The most popular media query features as a percent of pages.

Common breakpoints

The most common breakpoint in use across desktop and mobile devices is a `min-width` of 768px. 54% of sites use this breakpoint, closely followed by a `max-width` of 767px at 50%. The Bootstrap framework uses a `min-width` of 768px as its “Medium” size, so this may be the source of much of the usage. The other two high-ranking `min-width` values of 1200px (40%) and 992px (37%) are also found in Bootstrap.

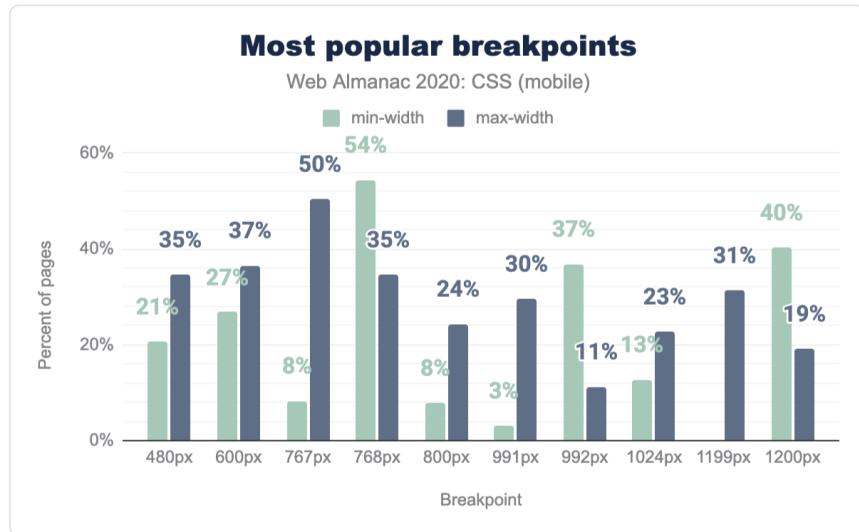


Figure 1.42. The most popular breakpoints by `min-width` and `max-width` as a percent of mobile pages.

Pixels are very much the unit that is used for breakpoints. There are a few instances of ems a long way down the list, however setting breakpoints in pixels appears to be the popular choice. There are probably many reasons for this. Legacy. All of the early articles on responsive design use pixels, and many people still think about targeting particular devices when creating responsive designs. Sizing using ems involves considering the size of the content rather than the device, and this is a newer way of thinking about web design, perhaps one yet to fully be taken advantage of along with intrinsic sizing methods for layout.

Properties used inside media queries

On mobile devices 79% and on desktop 77% of media queries are used to change the `display` property. Perhaps indicating that people are testing before switching to a flex or grid formatting context. Again, this may be linked frameworks, for example the Bootstrap responsive utilities. 78% of authors change the `width` property inside media queries, `margin`, `padding` and `font-size` all rank highly for changed properties.

Most popular properties used in media queries

Web Almanac 2020: CSS

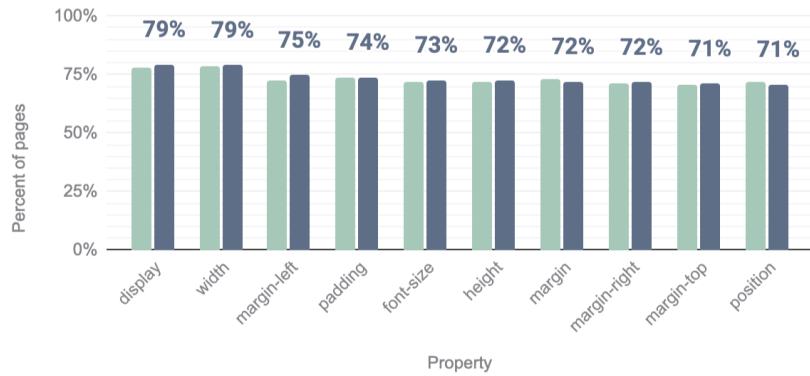
desktop
mobile


Figure 1.43. The most popular properties used in media queries as a percent of pages.

Custom properties

Last year, only 5% of websites were using custom properties. This year, adoption has skyrocketed. Using last year's query (which only counted declarations that set custom properties), usage has quadrupled on mobile (19.29%) and tripled on desktop (14.47%). However, when we look at values that reference custom properties via `var()`, we get an even better picture: 27% of mobile pages and 22% of desktop pages were using the `var()` function at least once, which indicates there is a sizeable number of pages only using `var()` to offer customization hooks, without ever setting a custom property.

While at first glance this is impressive adoption, it appears that a major driver is WordPress, as evidenced by the most popular custom property names, the top 4 of which ship with WordPress.

Naming

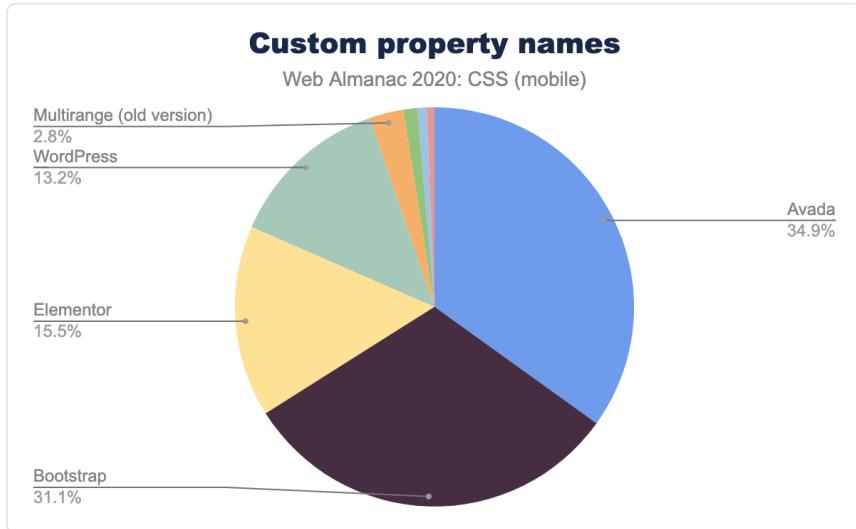


Figure 1.44. Relative popularity of custom property names per software entity as a percent of occurrences on mobile pages.

Out of the 1,000 top property names, fewer than 13 are ‘custom’, as in made up by individual web developers. The vast majority are associated with popular software, such as WordPress, Elementor, and Avada. To determine this, we took into account not only which custom properties appear in what software (by searching on GitHub), but also which properties appear in groups with similar frequencies. This does not necessarily mean that the main way a custom property ends up on a website is through usage of that software (people do still copy and paste!), but it does indicate there aren’t many organic commonalities between the custom properties that developers define. The only custom property names that seem to have organically made the list of top 1000 are `--height`, `--primary-color`, and `--caption-color`.

Usage by type

The biggest usage of custom properties appears to be naming colors and keeping colors consistent throughout. Approximately 1 in 5 desktop pages and 1 in 6 mobile pages uses custom properties in `background-color`, and the top 11 properties that contain `var()` references are either color properties or shorthands that contain colors. `Lengths` is the second biggest usage, with `width` and `height` being used with `var()` in 7% of mobile pages (interestingly, only around 3% of desktop pages). This is also confirmed by the types of most

popular values, with color values accounting for 52% of all custom property declarations. Dimensions (= a number + a unit, e.g. lengths, angles, times etc) were the second more popular type, higher than unitless numbers (12%), despite guidance to prefer the latter, since numbers can be converted to dimensions via `calc()` and multiplication, but dimensions cannot be converted to numbers as dividing with dimensions is not supported yet.

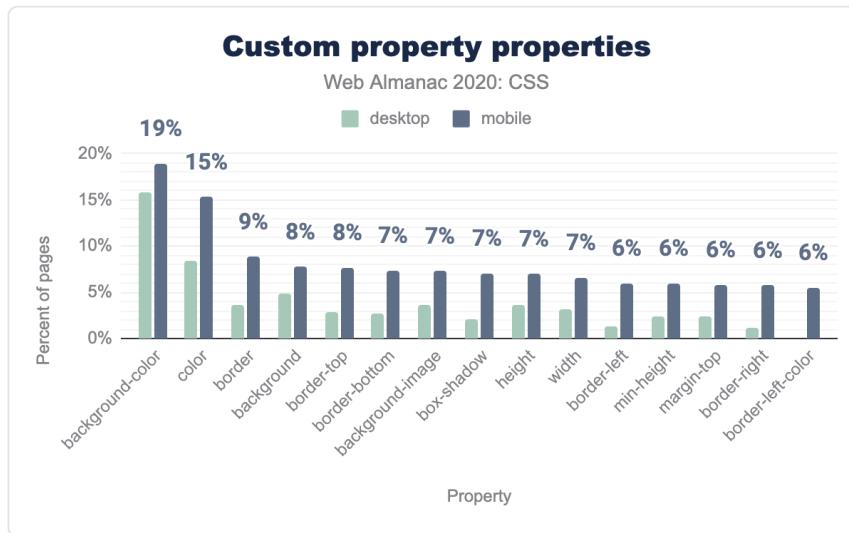


Figure 1.45. The most popular property names used with custom properties as a percent of pages.

In preprocessors, color variables are often manipulated to generate color variations, such as different tints. However, in CSS color modification functions are merely an unimplemented draft. Right now, the only way to generate new colors from variables is to use variables for individual components and plug them into color functions, such as `rgba()` and `hsla()`. However, fewer than 4% of mobile pages and 0.6% of desktop pages do that, indicating that the high usage of color variables is primarily to hold entire colors, with variations thereof being separate variables instead of dynamically generated.

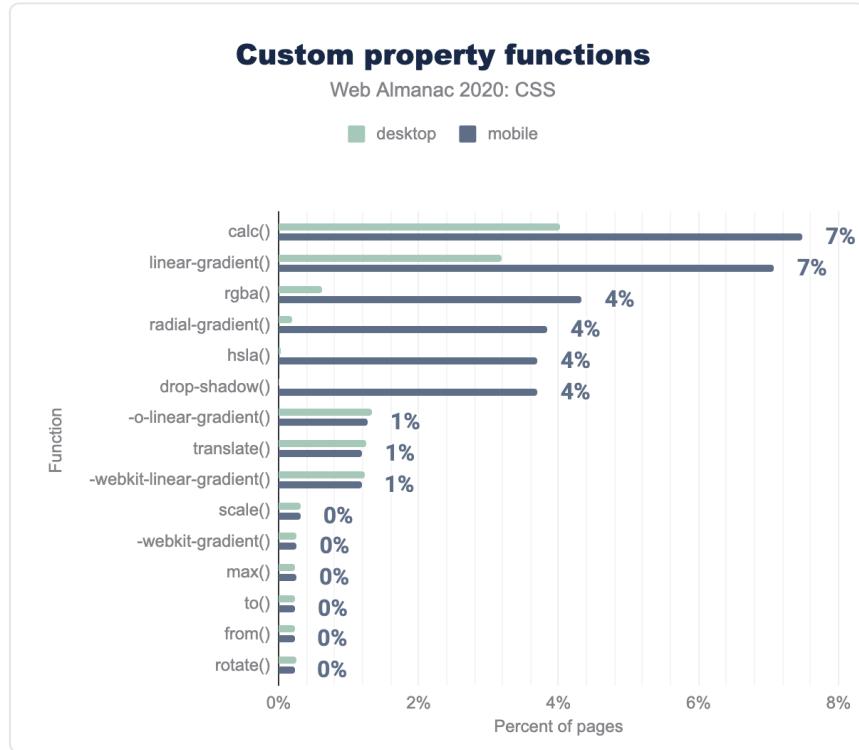


Figure 1.46. The most popular function names used with custom properties as a percent of pages.

Complexity

Next, we looked at how complex custom property usage is. One way to assess code complexity in software engineering is the shape of the dependency graph. We first looked at the *depth* of each custom property. A custom property set to a literal value like e.g. `#ffff` has a depth of 0, whereas a property referencing that via `var()` would have a depth of 1 and so on. For example:

```
:root {
  --base-hue: 335; /* depth = 0 */
  --base-color: hsl(var(--base-hue) 90% 50%); /* depth = 1 */
  --background: linear-gradient(var(--base-color), black); /* depth = 2 */
}
```

2 out of 3 custom properties examined (67%) had a depth of 0, and 30% had a depth of 1 (slightly less on mobile). Less than 1.8% had a depth of 2, and virtually none (0.7%) had a depth of 3+, which indicates rather basic usage. The upside of such basic usage is that it's harder to make mistakes: fewer than 0.5% of pages included cycles.

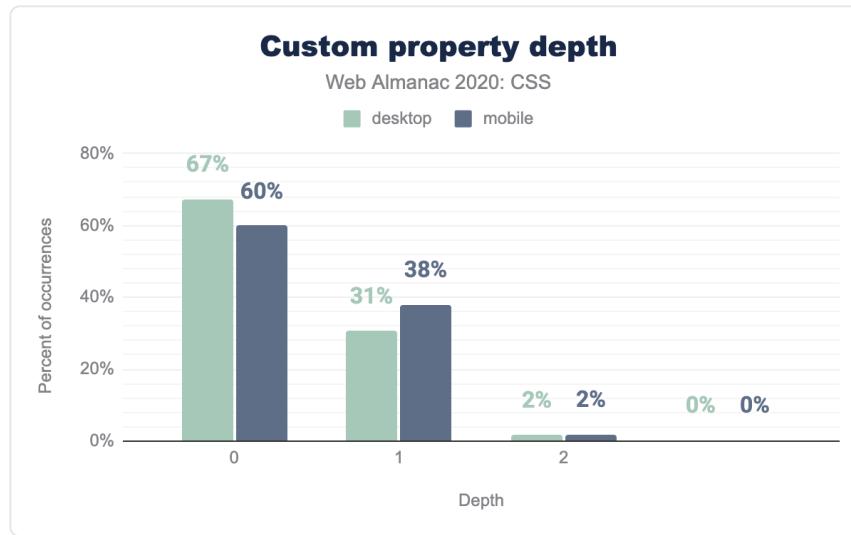


Figure 1.47. Distribution of depths of custom properties as a percent of occurrences.

Examining the selectors on which custom properties are declared further confirms that most custom property usage in the wild is fairly basic. Two out of three custom property declarations are on the root element, indicating that they are used essentially as global constants. It is important to note that many popular polyfills have required them to be global in this vein, so developers using said polyfills may not have had a choice.

CSS and JS

Houdini

You have likely heard of Houdini by now. Since several Houdini specs have shipped in browsers, we figured it's time to see if they are actually used in the wild yet. Short answer: no. And now for the longer answer...

First, we looked at the Properties & Values API, which allows developers to register a custom property and give it a type, an initial value, and prevent it from being inherited. One of the primary use cases is being able to animate custom properties, so we also looked at how

frequently custom properties are being animated.

As is common with bleeding edge tech, especially when not supported in all browsers, adoption in the wild has been extremely low. Only 32 desktop and 20 mobile pages were found to have any registered custom properties, though this excludes custom properties that were registered but were not being applied at the time of the crawl. Only 325 mobile pages and 330 desktop ones (0.00%) use custom properties in animations, and most (74%) of that seems to be driven by a Vue component. Virtually none of those appear to have registered them, though this is likely because the animation wasn't active at the time of the crawl, so there was no computed style needing to be registered.

The Paint API is a more broadly implemented Houdini spec which allows developers to create custom CSS functions that return `<image>` values, e.g. to implement custom gradients or patterns. Only 12 pages were found to be using `paint()`. Each worklet name (`hexagon`, `ruler`, `lozenge`, `image-cross`, `grid`, `dashed-line`, `ripple`) only appeared on one page each, so it appears the only in-the-wild use cases were likely demos.

Typed OM, another Houdini specification, allows access to structured values instead of the strings of the classic CSS OM. It appears to have considerably higher adoption compared to other Houdini specs, though still low overall. It is used in 9,864 desktop pages (0.18%) and 6,391 mobile ones (0.1%). While this may seem low, to put it in perspective, these are similar numbers to the adoption of `<input type="date">`! Note that unlike most stats in this chapter, these numbers reflect actual usage, and not just inclusion in a website's assets.

CSS-in-JS

There is so much discussion (or argument) about CSS-in-JS that one could assume everyone and their dog is using it. However, when we looked at usage of various CSS-in-JS libraries, it turned out that only about 2% of websites use *any* CSS-in-JS method, with Styled Components accounting for almost half of that.

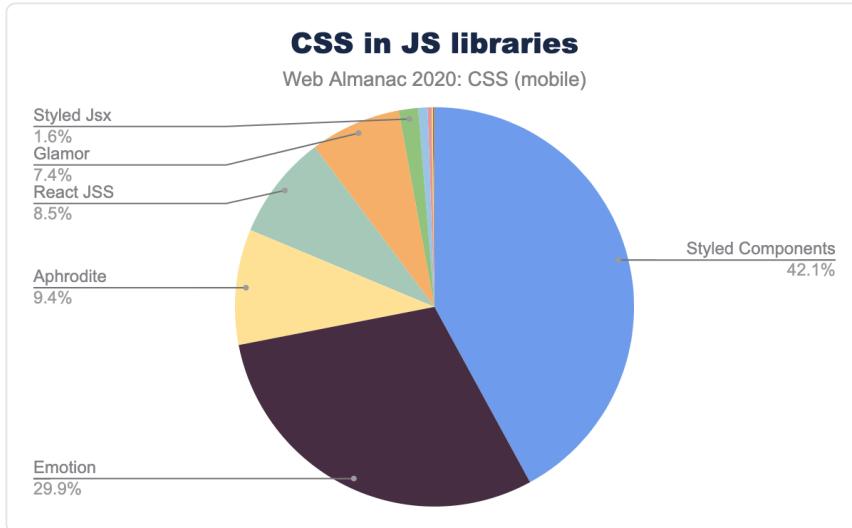


Figure 1.48. Relative popularity of CSS-in-JS libraries as a percent of occurrences on mobile pages.

Internationalization

Direction

When text is presented in horizontal lines, most writing systems display characters from left to right. Urdu, Arabic and Hebrew display characters from right to left, except for numbers, which are written from left to right; they are bidirectional. Some characters – such as brackets, quote marks, punctuation – could be used in either a left to right or a right to left context, and are said to be directionally neutral. Things get more complex when text strings of different languages are nested in one another - English text containing a short quote in Hebrew which contains some English words, for example. The Unicode bidirectional algorithm defines how to lay out paragraphs of mixed-direction text, but it needs to know the base direction of the paragraph.

To support bidirectionality, explicit support for indicating direction is available in both HTML via (the `dir` attribute and the `<bdo>` element), and CSS (the `direction` and `unicode-bidi` properties. We looked at usage of both HTML and CSS methods. Only 12.14% of pages on mobile (and a similar 10.76% on desktop) set the `dir` attribute on the `<html>` element.

Which is fine: most writing systems in the world are `ltr`, and the default `dir` value is `ltr`. Of the pages which did set `dir` on `<html>`, 91% set it to `ltr` while 8.5% set it to `rtl` and 0.32% to `auto` (the explicit direction is unknown value, mainly useful for templates which will be filled with unknown content). An even smaller number, 2.63%, set `dir` on `<body>`. Which is good,

because setting it on `<html>` also covers you for content in the `<head>`, like `<title>`.

Why set direction using HTML attributes rather than CSS styling? One reason is separation of concerns: direction has to do with content which is the purview of HTML. It is also the recommended practice: “*Avoid using CSS or Unicode control codes for managing direction where you can use markup*”. After all, the stylesheet might not load and the text still needs to be readable.

Logical vs physical properties

Many of the first properties we are taught when we learn CSS, things like `width`, `height`, `margin-left`, `padding-bottom`, `right` and so on are grounded on a specific physical direction. However, when content needs to be presented in multiple languages with different directionality characteristics, these physical directions are often language dependent, e.g. `margin-left` often needs to become `margin-right` in a right-to-left language such as Arabic. Directionality is a 2D characteristic. For example, `height` may need to become `width` when we are presenting content in vertical writing (such as traditional Chinese).

In the past, the only solution to these problems was a separate stylesheet with overrides for different writing systems. However, more recently CSS has acquired *logical* properties and values that work just like their *physical* counterparts but are sensitive to the directionality of their context. For example, instead of `width` we could write `inline-size`, and instead of `left` we could use the `inset-inline` property. In addition to logical *properties*, there are also logical *keywords*, such as `float: inline-start` instead of `float: left`.

While these properties are fairly well supported (with some exceptions), they are not used very much outside of user agent stylesheets. None of the logical properties were used on more than 0.6% of pages. Most usage was to specify margins and paddings. Logical keywords for `text-align` were used on 2.25% of pages, but apart from that, none of the other keywords were even encountered at all. This is by large driven by browser support: `text-align: start` and `end` have fairly good browser support whereas logical keywords for `clear` and `float` are only supported in Firefox.

Browser support

Vendor prefixes

Even though prefixing is now recognized as a failed way to introduce experimental features to developers, and browsers have largely stopped using it, opting for flags instead, a whopping 91% of pages still use at least one prefixed feature.

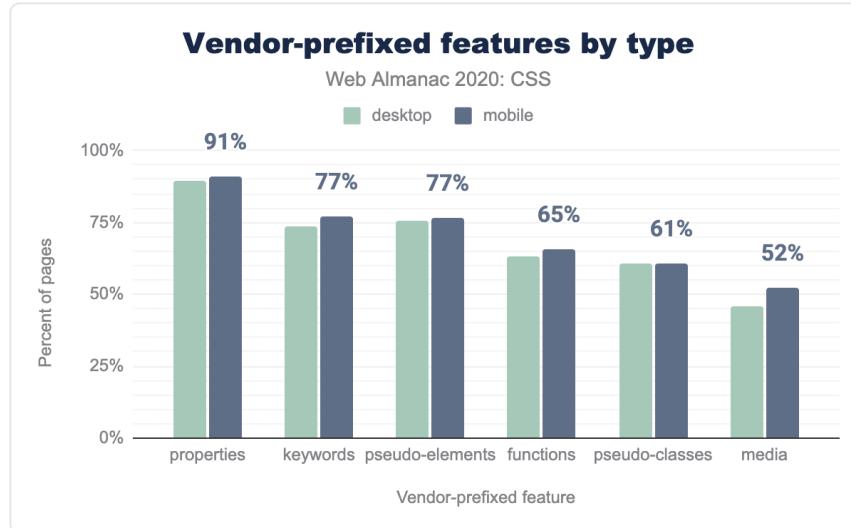


Figure 1.49. The most popular vendor-prefixed features by type as a percent of pages.

91.05%

Figure 1.50. Percent of mobile pages using any vendor prefixed feature.

Prefixed properties take up the lion's share of that, since **84% of all prefixed features used were properties**. This is most likely a remnant of the prefix-happy CSS3 era circa 2009-2014. This is also evident from the most popular prefixed ones, none of which have actually needed prefixes since 2014:

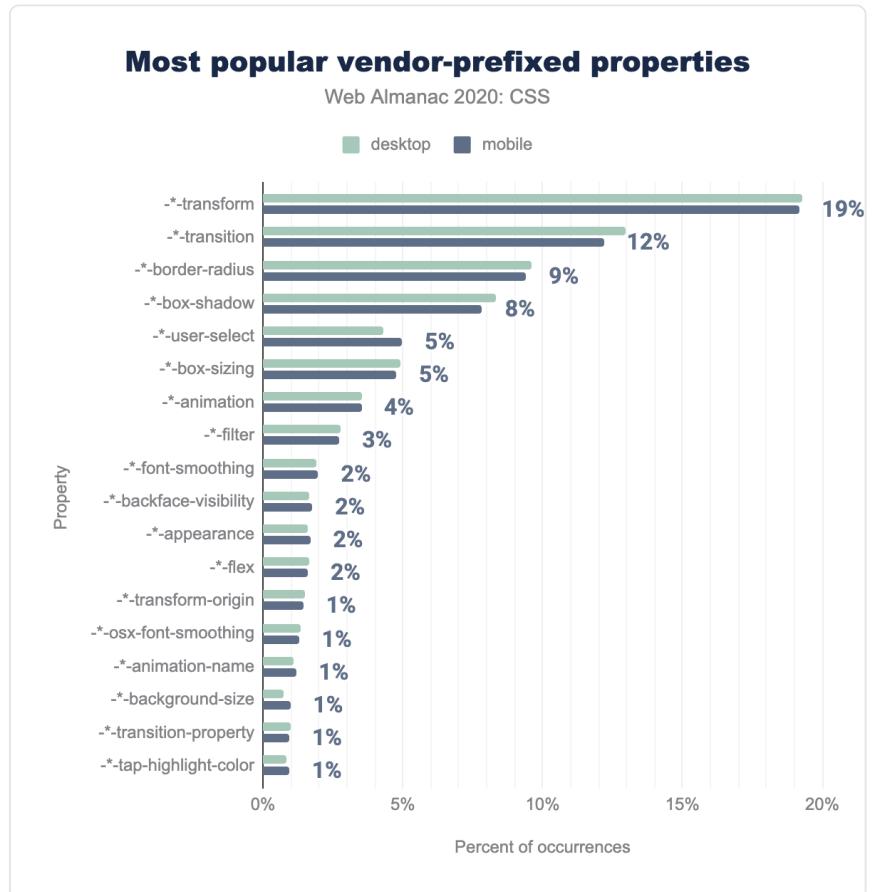


Figure 1.51. Relative popularity of properties that are most used with vendor prefixes, as a percent of occurrences.

Some of these prefixes, like `-moz-border-radius`, haven't been useful since 2011. Even more mind-boggling, other prefixed properties that never existed, are still moderately common, with roughly 9% of all pages including `-o-border-radius`!

It may come as no surprise that `-webkit-` is by far the most popular prefix, with half of prefixed properties using it:

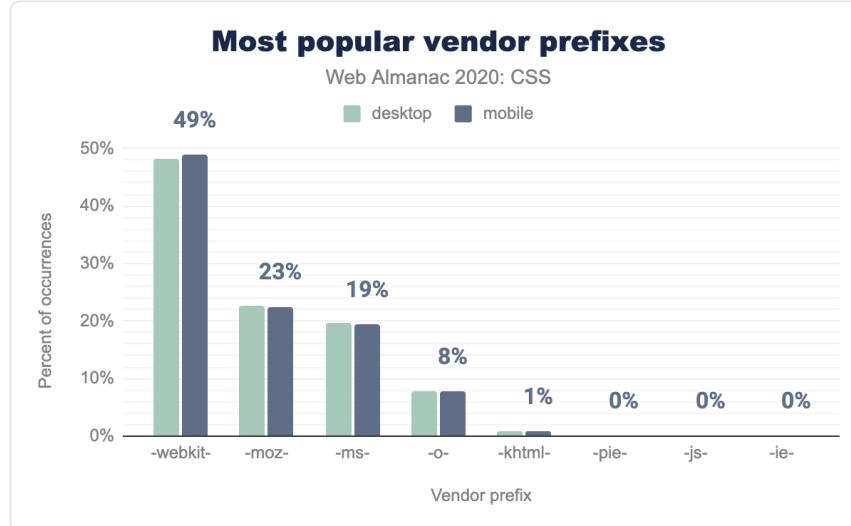


Figure 1.52. Relative popularity of vendor prefixes, as a percent of occurrences.

Prefixed pseudo-classes are not nearly as common as properties, with none of them being used in more than 10% of pages. Nearly two thirds of all prefixed pseudo-classes **overall** are for styling placeholders. In contrast, the standard `:placeholder-shown` pseudo-class is barely used, encountered in less than 0.34% of pages.

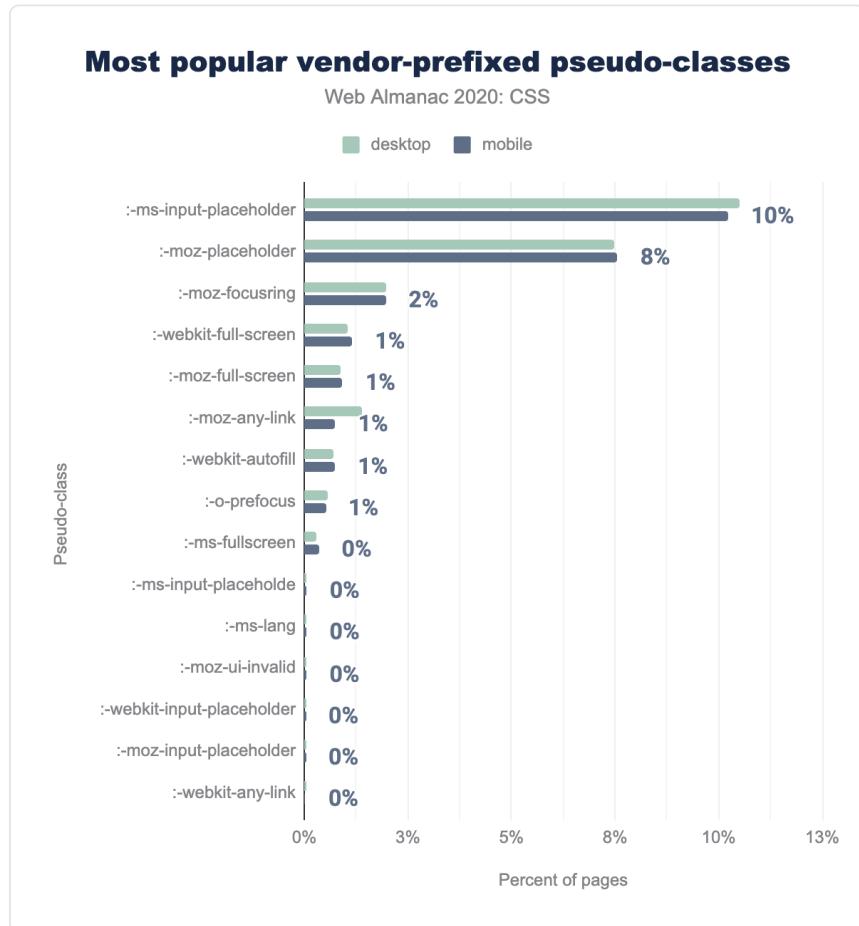


Figure 1.53. The most popular vendor-prefixed pseudo-classes as a percent of pages.

The most common prefixed pseudo-element is `::-moz-focus-inner`, used to disable Firefox's inner focus ring. It makes up almost a quarter of prefixed pseudo-elements and for which there is no standard alternative. Another quarter of prefixed pseudo-elements is yet again for styling placeholders, while the standard version, `::placeholder`, trails far behind, used in only 4% of pages.

The remaining half of prefixed pseudo-elements is primarily devoted to styling scrollbars and Shadow DOM of native elements (search inputs, video & audio controls, spinner buttons, sliders, meters, progress bars). This indicates a strong developer need for customization of built-in UI controls, for which standards-compliant CSS still falls short, although there are multiple ongoing CSS WG discussions to ameliorate that.

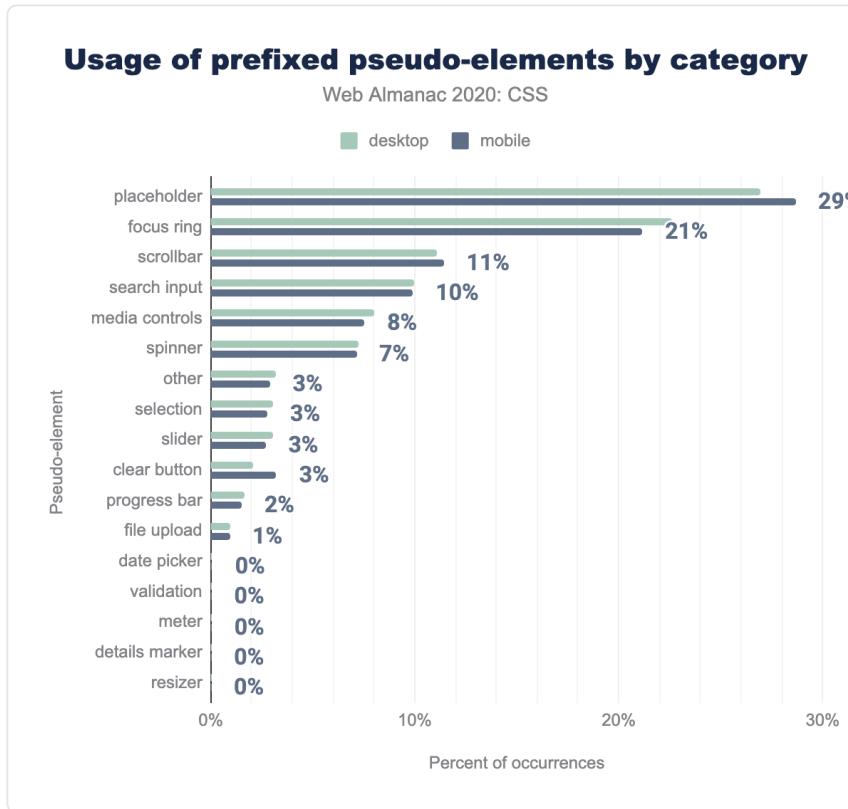


Figure 1.54. Relative popularity of vendor-prefixed pseudo-elements by purpose as a percent of occurrences.

It is no secret that Chrome and Safari have been way more prefix-happy, but it is especially true with pseudo-elements: nearly half of all prefixed pseudo-elements we found had a `-webkit-` prefix.

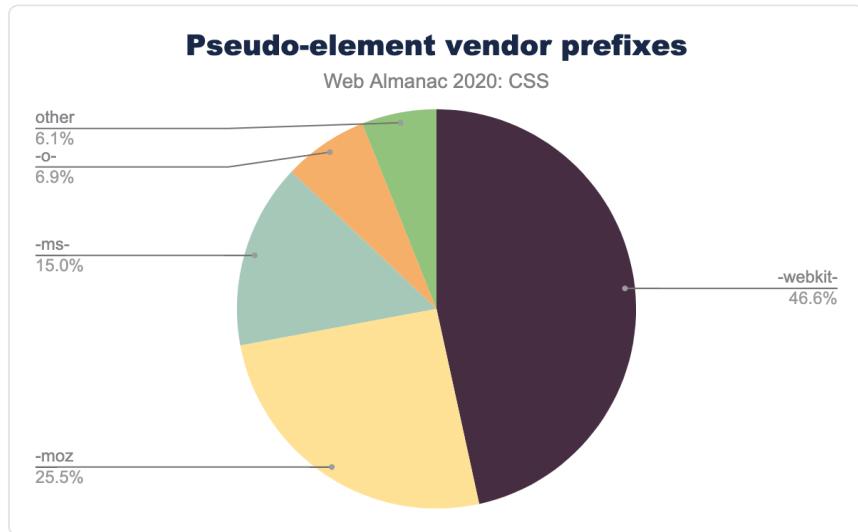


Figure 1.55. Relative popularity of pseudo-element vendor prefixes as a percent of occurrences on mobile pages.

Nearly all usage of prefixed functions (98%) is to specify gradients, even though this has not been necessary since 2014. The most popular of these is `-webkit-linear-gradient()` used in over a quarter of pages examined. The remaining <2% is primarily for calc, for which a prefix has not been necessary since 2013.

98.22%

Figure 1.56. Percent of gradient functions across all occurrences of vendor-prefixed functions in mobile pages

Usage of prefixed media features is lower overall, with the most popular one, `-webkit-min-pixel-ratio` used in 13% of pages to detect "Retina" displays. The corresponding standard media feature, `resolution` has finally surpassed it in popularity and is used in 1,373,813 mobile pages (22%) and 811,411 desktop pages (15%).

Overall, `-* -min-pixel-ratio` comprises three quarters of prefixed media features on desktop and about half on mobile. The reason for the difference is not reduced mobile usage, but that another prefixed media feature, `-* -high-contrast`, is far more popular on mobile making up almost the entire other half of prefixed media features, but only 18% on desktop.

The corresponding standard media feature, forced-colors is still experimental and behind a flag in Chrome and Firefox, and thus, did not appear at all in our analysis.

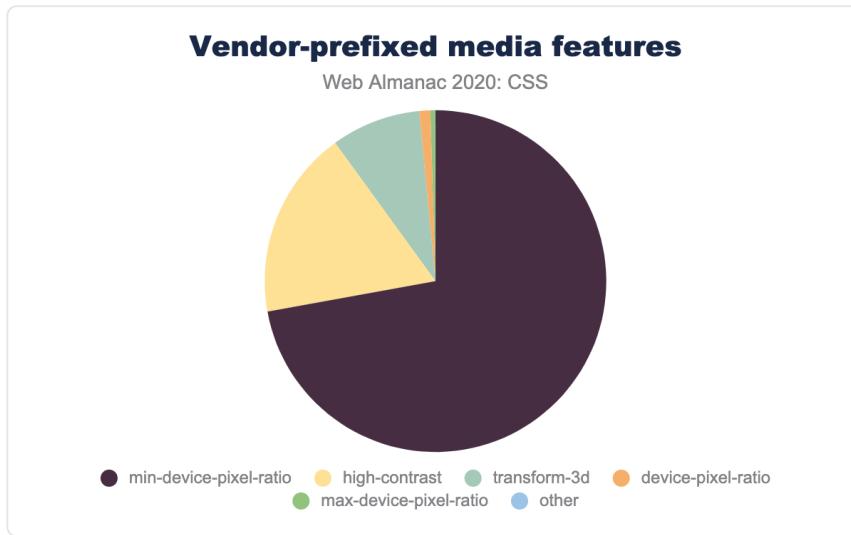


Figure 1.57. Relative popularity of vendor-prefixed media features as a percent of occurrences on mobile pages.

Feature queries

Feature queries (@supports) have been steadily gaining traction for the past few years, and were used in 39% of pages, a notable increase from last year's 30%.

But what are they used for? We looked at the most popular queries. The results may come as a big surprise — it was to us! We expected grid-related queries to top the list, but instead, the most popular feature queries by far are for `position: sticky`! They comprise half of all feature queries and are used in about a quarter of pages. In contrast, grid-related queries account for only 2% of all queries, indicating that developers feel comfortable enough with Grid's browser support that they don't need to use it only as progressive enhancement.

What is even more mysterious is that `position: sticky` itself is not used as much as the feature queries about it, appearing in 10% of desktop pages and 13% of mobile pages. So there are over half a million pages that detect `position: sticky` without ever using it! Why?!

Lastly, it was encouraging to see `max()` already in the top 10 most detected features, appearing in 0.6% of desktop pages and 0.7% of mobile pages. Given that `max()` (and `min()`, and `clamp()`) was only supported across the board this year, it is quite impressive adoption

and highlights how desperately developers needed this.

A small but notable number of pages (around 3000 or 0.05%) were oddly using `@supports` with CSS 2 syntax, such as `display: block` or `padding: 0px`, syntax that existed well before `@supports` was implemented. It is unclear what this was meant to achieve. Perhaps it was used to shield CSS rules from old browsers that don't implement `@supports`?

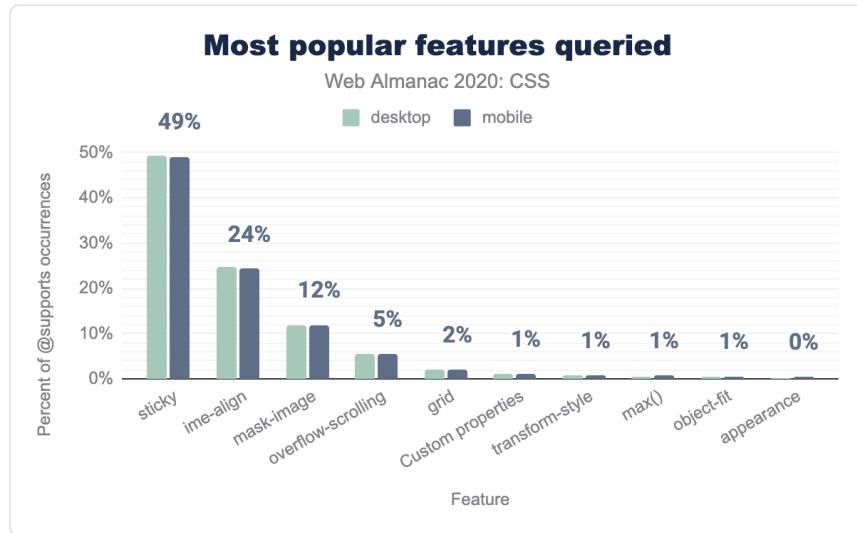


Figure 1.58. Relative popularity of `@supports` features queried as a percent of occurrences.

Meta

Declaration repetition

To tell how efficient and maintainable a stylesheet is, one rough factor is declaration repetition, that is, the ratio between unique (different) and total number of declarations. The factor is a rough one because it's not trivial to normalize declarations (`border: none`, `border: 0`, even `border-width: 0` —plus a few more—are all different but say the same thing), and also because there are levels for the repetition: media query (most useful but harder to measure), style sheet, or data set level as with the Almanac's overall metrics.

We did look at declaration repetition and found that the median web page, on mobile, uses a total of 5,454 declarations, of which 2,398 are unique. The median ratio (which is based on the data set, not these two values) comes out at 45.43%. What this means is that on the median

page, each declaration is used roughly two times.

Percentile	Unique / Total
10	30.97%
50	45.43%
90	63.67%

Figure 1.59. Distribution of repetition ratios on mobile pages.

These ratios are better, then, than what we know from scarce previous data. In 2017, Jens Oliver Meiert sampled 220 popular websites and came out with the following averages: 6,121 declarations, of which 1,698 were unique, and a unique/total ratio of 28% (median 34%). The topic could need further investigation, but from the little we know so far, declaration repetition is tangible—and may have either improved, or be more of a problem for the more popular and likely larger sites.

Shorthands and longhands

Some shorthands are more successful than others. Sometimes the shorthand is sufficiently easy to use and its syntax memorable, that we end up only using the longhands intentionally, when we want to override certain values independently. And then there are these shorthands that are hardly ever used because their syntax is too confusing.

Shorthands before longhands

Some shorthands are more successful than others. Sometimes the shorthand is sufficiently easy to use and its syntax memorable, that we end up only using the longhands intentionally, when we want to override certain values independently. And then there are these shorthands that are hardly ever used because their syntax is too confusing. Shorthands before longhands Using a shorthand and overriding it with a few longhands in the same rule is a good strategy for a variety of reasons:

First, it's good defensive coding. The shorthand resets all its longhands to their initial values if they have not been explicitly specified. This prevents rogue values coming in through the cascade.

Second, it's good for maintainability, to avoid repetition of values when the shorthand has smart defaults. For example, instead of `margin: 1em 1em 0 1em` we can write:

```
margin: 1em;  
margin-bottom: 0;
```

Similarly, for list-valued properties, longhands can help us reduce repetition when a value is the same across all list values:

```
background: url("one.png"), url("two.png"), url("three.png");  
background-repeat: no-repeat;
```

Third, for cases where parts of the shorthand's syntax are too weird, longhands can help improve readability:

```
/* Instead of: */  
background: url("one.svg") center / 50% 50% content-box border-box;  
  
/* This is more readable: */  
background: url("one.svg") center;  
background-size: 50% 50%;  
background-origin: content-box;  
background-clip: border-box;
```

So how frequently does this occur? Very, as it turns out. 88% of pages use this strategy at least once. By far, the most frequent longhand this happens with is `background-size`, accounting for 40% of all longhands that come after their shorthand, indicating that the slash syntax for `background-size` in `background` may not have been the most readable or memorable syntax we could have come up with. No other longhand comes close to this frequency. The remaining 60% is a long tail spread across many other properties evenly.

Most popular longhand properties after shorthands

Web Almanac 2020: CSS

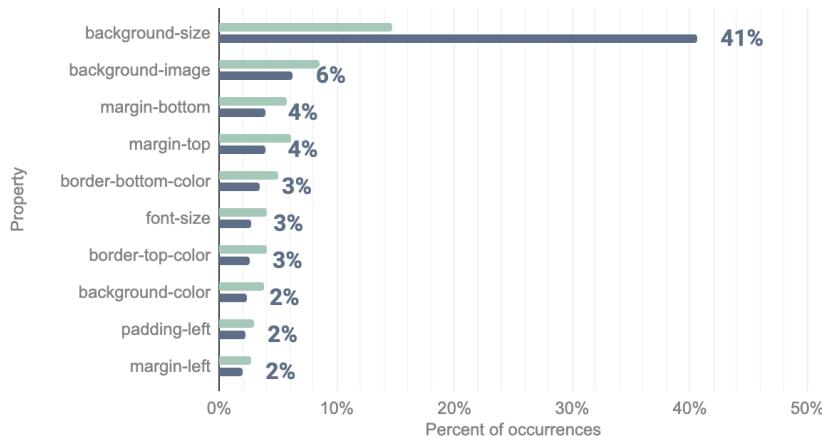
■ desktop ■ mobile

Figure 1.60. Most popular longhands that come after their shorthands in the same rule.

font

The `font` shorthand is fairly popular (used 49 million times on 80% of pages) but used far less than most of its longhands (except `font-variant` and `font-stretch`). This indicates that most developers are comfortable using it (since it appears on so many websites). Developers often need to override specific typographic aspects on descendant rules, which likely explains why the longhands are used so much more.

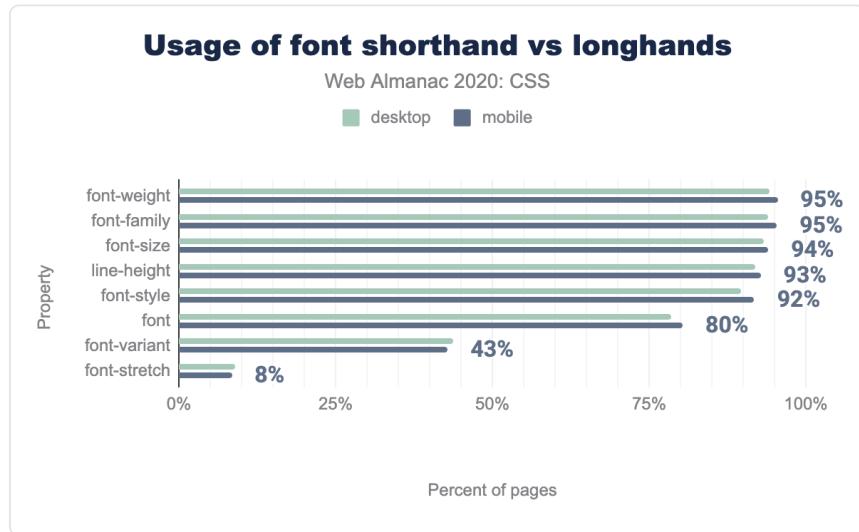


Figure 1.61. Adoption of `font` shorthand and longhand properties.

background

As one of the oldest shorthands, `background` is also highly used, appearing 1 billion times in 92% of pages. It's used more frequently than any of its longhands except `background-color` which is used 1.5 billion times, in roughly the same number of pages. However, this doesn't mean developers are fully comfortable with all of its syntax: nearly all (>90%) of `background` usage is very simple, with one or two values, most likely colors and images or images and positions. For anything further, the longhands are seen as more self-explanatory.

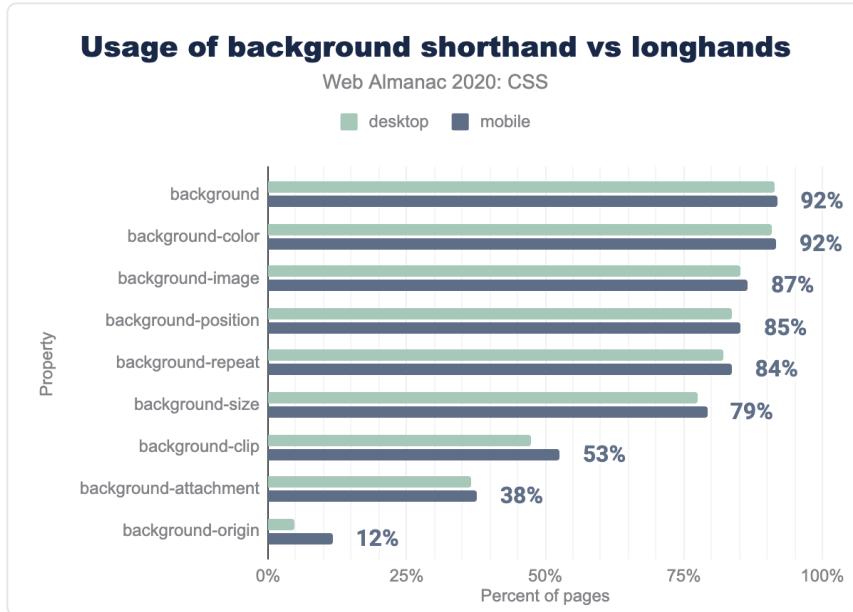


Figure 1.62. Usage comparison of the `background` shorthand and its longhands.

Margins and paddings

Both the `margin` and `padding` shorthands, as well as their longhands were some of the most highly used CSS properties. Padding is considerably more likely to be specified as a shorthand (1.5B uses for `padding` vs 300-400M for each shorthand), whereas there is less of a difference for margin (1.1B uses of `margin` vs 500-800M for each of its longhands). Given the initial confusion of many CSS developers about the clockwise order of values in these shorthands and the repetition rule for 2 or 3 values, we expected that most of these uses of the shorthands would be simple (1 value), however we saw the entire range of 1,2,3 or 4 values. Obviously 1 or 2 values were more common, but 3 or 4 were not at all uncommon, occurring in over 25% of `margin` uses and over 10% of `padding` usage.

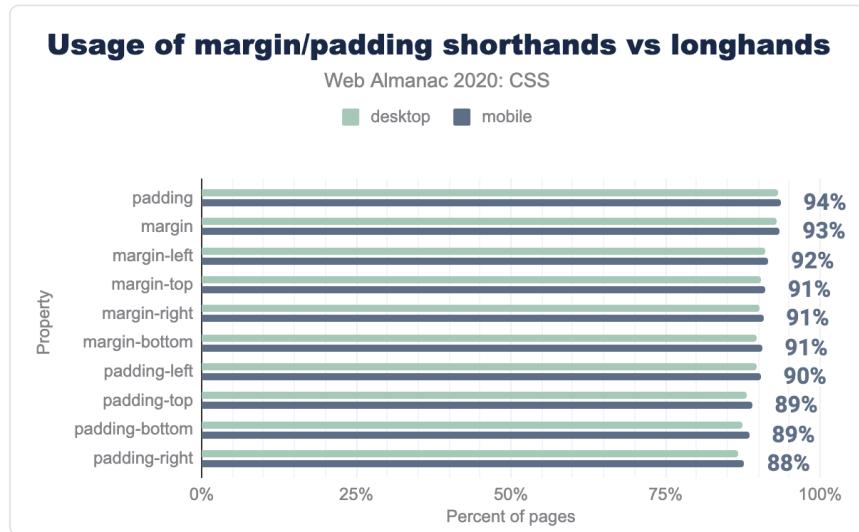


Figure 1.63. Usage comparison of the `margin` & `padding` shorthands and their longhands.

Flex

Nearly all `flex`, `flex-*` properties are very highly used, appearing in 30-60% of pages. However, both `flex-wrap` and `flex-direction` are used far more than their shorthand, `flex-flow`. When `flex-flow` is used, it is used with two values, i.e. as a shorter way to set both of its longhands. Despite the elaborate sensible defaults for using `flex` with one or two values, around 90% of usage consists of the 3 value syntax, explicitly setting all three of its longhands.

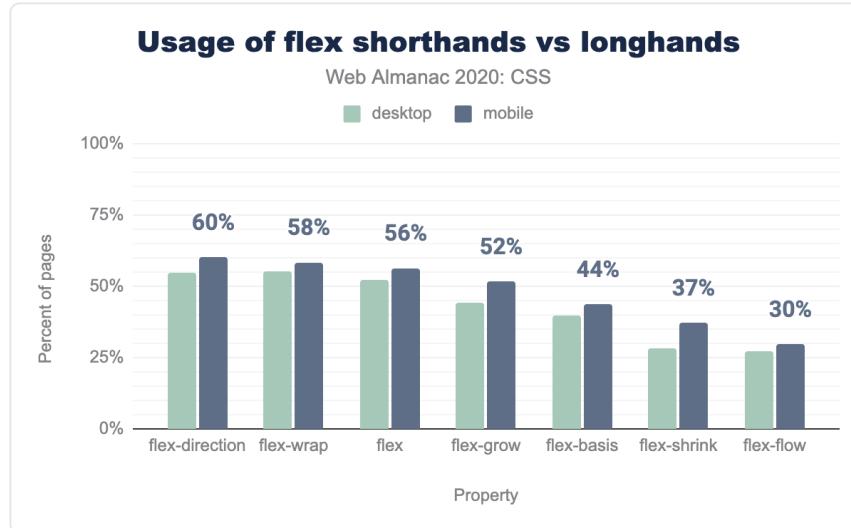


Figure 1.64. Usage comparison of the flex shorthands and their longhands.

Grid

Did you know that `grid-template-columns`, `grid-template-rows`, and `grid-template-areas` are actually shorthands of `grid-template`? Did you know that there's a `grid` property and all of those are some of its longhands? No? Well, you're in good company: neither do most developers. The `grid` property was only used in 5,279 websites (0.08%) and `grid-template` on 8,215 websites (0.13%). In comparison, `grid-template-columns` is used in 1.7 million websites, over 200 times more!

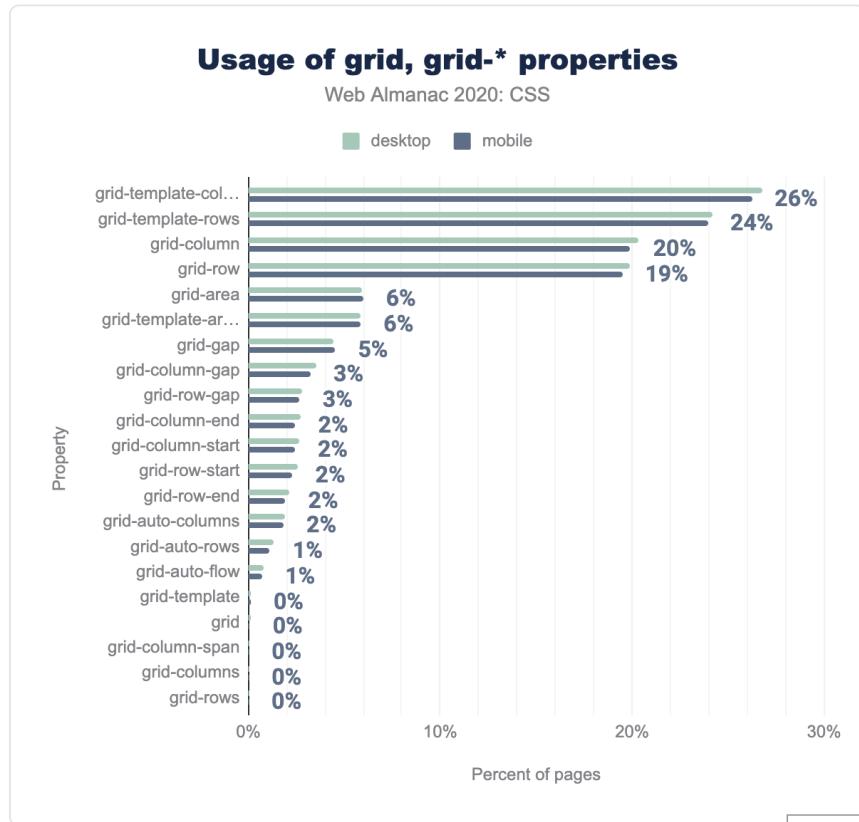


Figure 1.65. Usage comparison of the grid shorthands and their longhands.

CSS mistakes

Syntax errors

For most of the metrics in this chapter, we used Rework, a CSS parser. While this helps dramatically improve accuracy, it also means we could be less forgiving of syntax errors compared to a browser. Even if one declaration in the entire stylesheet has a syntax error, parsing would fail and that stylesheet would be left out of the analysis. But how many stylesheets do contain such syntax errors? Quite substantially more on desktop than mobile it turns out! More specifically, nearly 10% of stylesheets found on desktop pages included at least one unrecoverable syntax error, whereas only 2% of mobile. Do note that these are essentially lower bounds for syntax errors, since not all syntax errors actually cause parsing to fail. For example, a missing semicolon would just result in the next declaration being parsed as part of

the value (e.g. `{property: "color", value: "red background: yellow"}`), it would not cause the parser to fail.

Nonexistent properties

We also looked at most common nonexistent properties, by using a list of known properties. We excluded prefixed properties from this part of the analysis, and manually excluded unprefixed proprietary properties (e.g. IE's `behavior`, which oddly still appears on 200K websites). Out of the remaining nonexistent properties:

- 37% of them were a mangled form of a prefixed property (e.g. `webkit-transition` or `-transition`)
- 43% were an unprefixed form of a property that only exists only prefixed (e.g. `font-smoothing`, which appeared on 384K websites), probably included for compatibility under the incorrect assumption that it's standard, or due to wishful thinking that it will become standard.
- A typo that has found its way to a popular library. Through this analysis, we found that the property `white-wpace` was present in 234,027 websites. This is way too many websites for the same typo to have occurred organically, so we decided to look into it. And lo and behold, it turns out it was the Facebook widget! The fix is already in.
- And another oddity: The property `font - rendering` appears on 2,575 pages. However, we cannot find evidence of such a property existing, with or without a prefix. There is the nonstandard `-webkit-font-smoothing` which is wildly popular, appearing in 3 million websites, or about 49% of pages, but `font-rendering` is not sufficiently close to be a misspelling. There is `text-rendering` which is used in about 100K of websites, so it is conceivable that 2.5K developers all misremembered and coined a portmanteau of `font-smoothing` and `text-rendering`.

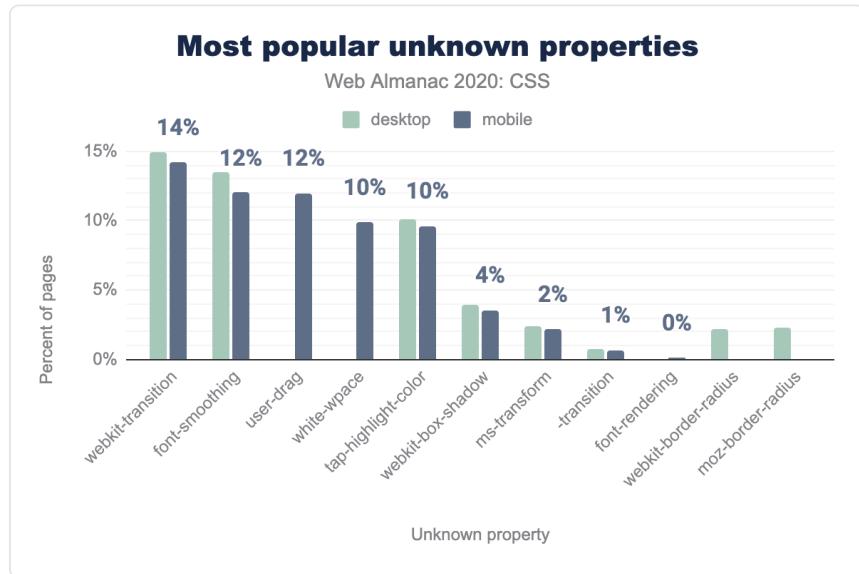


Figure 1.66. Most popular unknown properties.

Longhands before shorthands

Using longhands *after* shorthands is a nice way to use the defaults and override a few properties. It is especially useful with list-valued properties, where using a longhand helps us avoid repeating the same value multiple times. The **opposite** on the other hand — using longhands *before* shorthands — is always a mistake, since the shorthand will overwrite the longhand. For example, take a look at this:

```
background-color: rebeccapurple; /* longhand */
background: linear-gradient(white, transparent); /* shorthand */
```

This will not produce a gradient from `white` to `rebeccapurple`, but from `white` to `transparent`. The `rebeccapurple` background color will be overwritten by the `background` shorthand that comes after it that resets all its longhands to their initial values.

There are two main reasons that developers make this kind of mistake: either a misunderstanding about how shorthands work and which longhand is reset by which shorthand, or simply leftover cruft from moving declarations around.

So how common is this mistake? Surely, it cannot be *that* common in the top 6 million websites,

right? *Wrong*. It turns out, it is **exceedingly common**, occurring at least once in 54% of websites!

This kind of confusion seems to happen way more with the `background` shorthand than any other shorthand: over half (55%) of these mistakes involve putting `background-*` longhands before `background`. In this case, this may actually not be a mistake at all, but good progressive enhancement: Browsers that don't support a feature -- such as linear gradients -- will render the previously defined longhand values, in this case, a background color. Browsers that do understand the shorthand override the longhand value, either implicitly or explicitly.

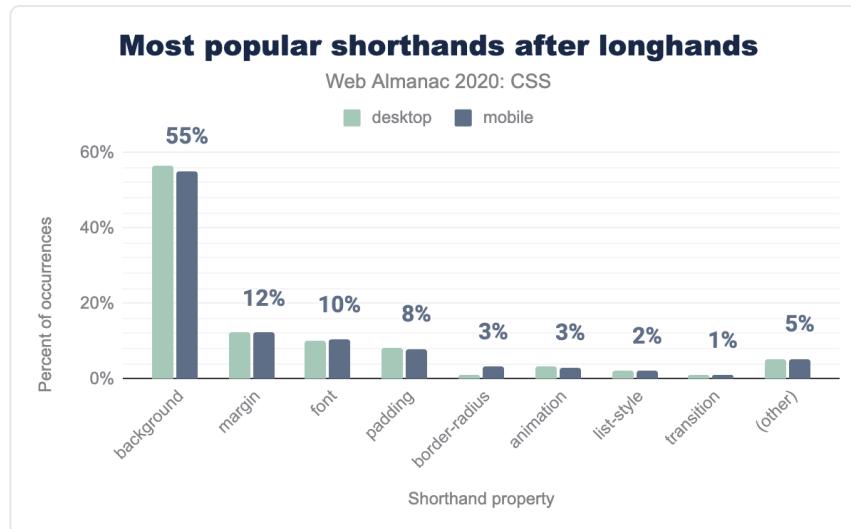


Figure 1.67. Most popular shorthands after longhands.

Sass

While analyzing CSS code tells us what CSS developers *are* doing, looking at preprocessor code can tell us a bit about what CSS developers *want* to be doing, but *can't*, which in some ways is more interesting. We used CSS files with sourcemaps to extract and analyze SCSS stylesheets in the wild. We chose to look at SCSS because it's the most popular preprocessing syntax, based on our analysis of sourcemaps.

We've known for a while that developers need color modification functions, and are working on them in CSS Color 5. However, analyzing SCSS function calls gives us hard data to prove just how necessary color modification functions are, and also tells us which types of color modifications are most commonly needed.

Overall, over one third of all Sass function calls are to modify colors or extract color components. Virtually all color modifications we found were rather simple. Half were to make colors darker. In fact, `darken()` was the most popular Sass function call overall, used even more than the generic `if()`! It appears that a common strategy is to define bright core colors, and use `darken()` to create darker variations. The opposite, making them lighter, is less common, with only 5% of function calls being `lighten()`, though that was still the 6th most popular function overall. Functions that change the alpha channel were about 4% of overall function calls and mixing colors makes up 3.5% of all function calls. Other types of color modifications such as adjusting hue, saturation, red/green/blue channels, or the more complex `adjust-color()` were used very sparingly.

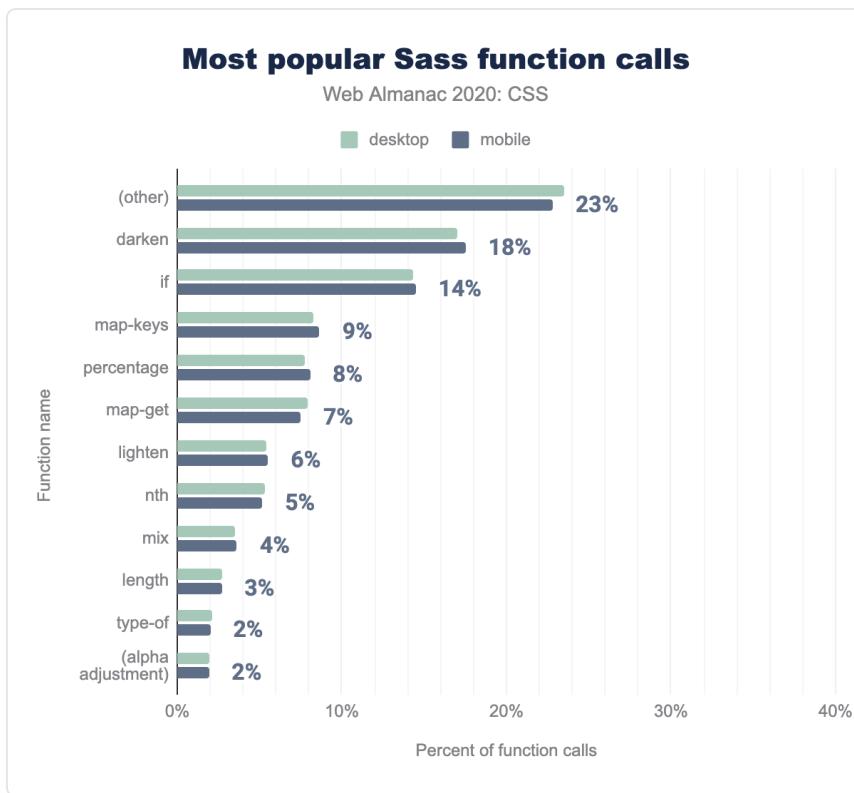


Figure 1.68. Most popular Sass function calls.

Defining custom functions is something that has been discussed for years in Houdini, but studying Sass stylesheets gives us data on how common the need is. Quite common, it turns out. At least **half** of SCSS stylesheets studied contain custom functions, since the median SCSS sheet contains not one, but two custom functions.

There are also recent discussions in the CSS WG about introducing a limited form of conditionals, and Sass gives us some data on how commonly this is needed. Almost two thirds of SCSS sheets contain at least one `@if` block, making up almost two thirds of all control flow statements. There is also an `if()` function for conditionals within values, which is the second most common function used overall (14%).

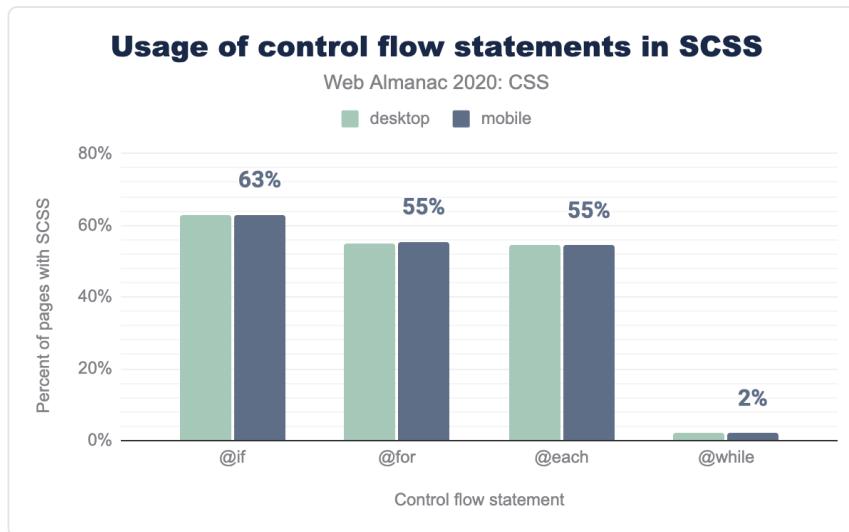


Figure 1.69. Usage of control flow statements in SCSS.

Another future spec that is currently worked on is CSS Nesting, which will afford us the ability to nest rules within other rules similarly to what we can do in Sass and other preprocessors by using `&`. How commonly is nesting used in SCSS sheets? Very, it turns out. The vast majority of SCSS sheets use at least one explicitly nested selector, with pseudo-classes (e.g. `&:hover`) and classes (e.g. `&.active`) making up three quarters of it. And this does not account for implicit nesting, where a descendant is assumed and the `&` character is not required.

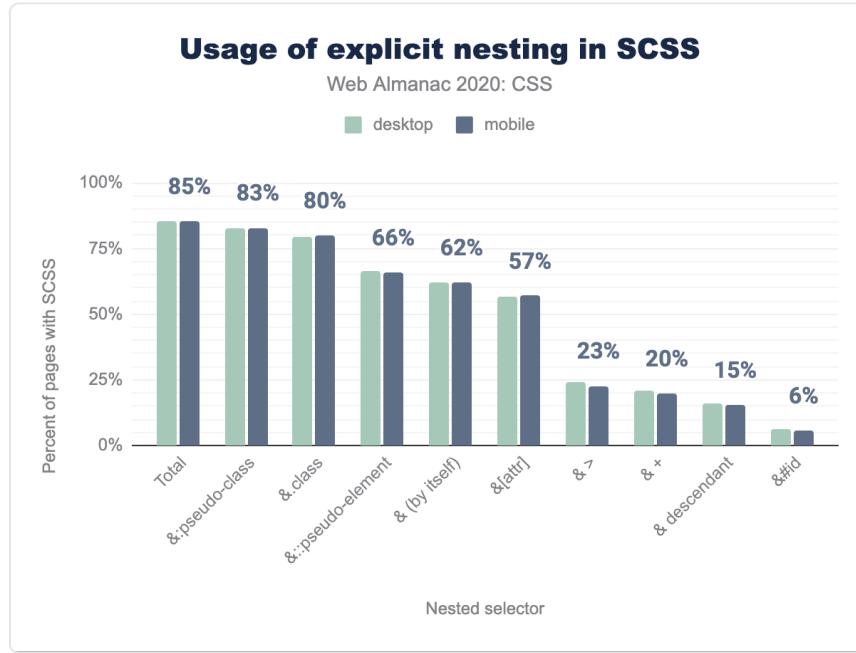


Figure 1.70. *usage-of-explicit-nesting-in-scss*.

Conclusion

Whew! That was a lot of data! We hope you have found it as interesting as we did, and perhaps even formed your own insights about some of them.

One of our takeaways was that popular libraries such as WordPress, Bootstrap, and Font Awesome are primary drivers behind adoption of new features, while individual developers tend to be more conservative.

Another observation is that there is more old code on the Web than new code. The Web in practice spans a huge range, from code that could have been written 20 years ago to bleeding edge tech that only works in the latest browsers. What this study showed us, though, is that there are powerful features that are often misunderstood and underused, despite good interoperability.

It also showed us some of the ways that developers want to use CSS but can't, and gave us some insight on what they find confusing. Some of this data will be brought back to the CSS WG to help drive CSS's evolution, because data-driven decisions are the best kind of decisions.

We are excited about the ways that this analysis could have further impact in the way we develop websites, and looking forward to seeing how these metrics develop over time!

Authors



Lea Verou

Twitter: @leaverou GitHub: LeaVerou URL: <https://leaverou.me/>

Lea teaches HCI & web programming¹ and researches how to make web programming easier² at MIT³. She is a bestselling technical author⁴ and experienced speaker⁵. She is passionate about open web standards, and is a longtime CSS Working Group⁶ member. Lea has started several popular open source projects and web applications⁷, such as Prism⁸, and Awesomplete⁹. She tweets @leaverou and blogs at leaverou.me¹⁰.



Chris Lilley

Twitter: @svgeesus GitHub: svgeesus URL: <https://svggees.us/>

Chris Lilley is a Technical Director at the World Wide Web Consortium (W3C). Considered “the father of SVG”, he also co-authored PNG, was co-editor of CSS2, chaired the group that developed @font-face, and co-developed WOFF. Ex Technical Architecture Group. Chris is still trying to get Color Management on the Web, sigh. Currently working on CSS levels 3/4/5 (no, really), Web Audio, and WOFF2..



Rachel Andrew

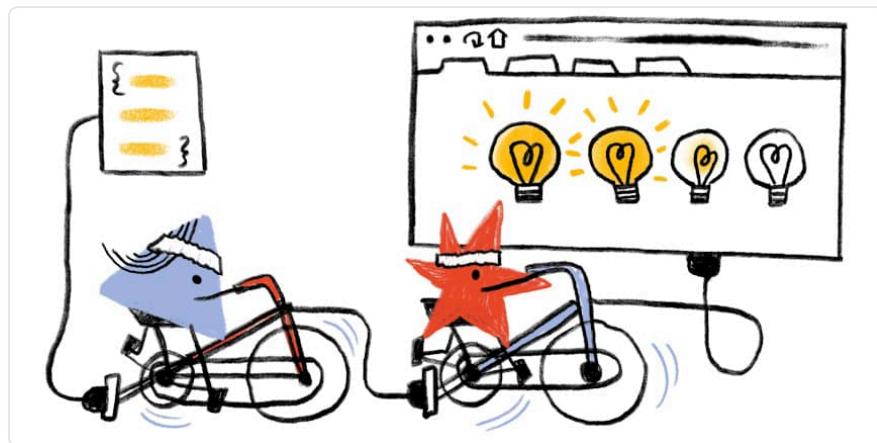
Twitter: @rachelandrew GitHub: rachelandrew URL: <https://rachelandrew.co.uk/>

I’m a web developer, writer, public speaker. Co-founder of Perch CMS¹¹ and Notist¹². Member of the CSS Working Group¹³. Editor in Chief of Smashing Magazine¹⁴.

-
1. <https://designftw.mit.edu>
 2. <https://mavio.io>
 3. <https://mit.edu>
 4. <https://www.amazon.com/CSS-Secrets-Lea-Verou/dp/1449372635?tag=leaverou-20>
 5. <https://leaverou.me/speaking>
 6. <https://www.w3.org/Style/CSS/members.en.php3>
 7. <https://github.com/leaverou>
 8. <https://prismjs.com>
 9. <https://github.com/leaverou/awesomplete>
 10. <https://leaverou.me>
 11. <https://grabaperch.com>
 12. <https://notist.st>
 13. <https://www.w3.org/wiki/CSSWG>
 14. <https://www.smashingmagazine.com/>

Part I Chapter 2

JavaScript **[UNEDITED]**



Written by Tim Kadlec

Reviewed by Sawood Alam and Artem Denysov

Analyzed by Rick Viscomi and Paul Calvano

Introduction

JavaScript has come a long way from its humble origins as the last of the three web cornerstones, alongside CSS and HTML. Today, JavaScript has started to infiltrate a broad spectrum of the technical stack. JavaScript is no longer confined to the client-side—it's an increasingly popular choice for build tools and server-side scripting, and is creeping its way into the CDN layer as well thanks to edge computing solutions.

Developers love us some JavaScript. According to the Markup chapter, the `script` element is the 6th most popular HTML element in use (ahead of elements like `p` and `i`, among countless others). We spend around 14 times as many bytes on it as we do on HTML, the building block of the web, and 6 times as many bytes as CSS.

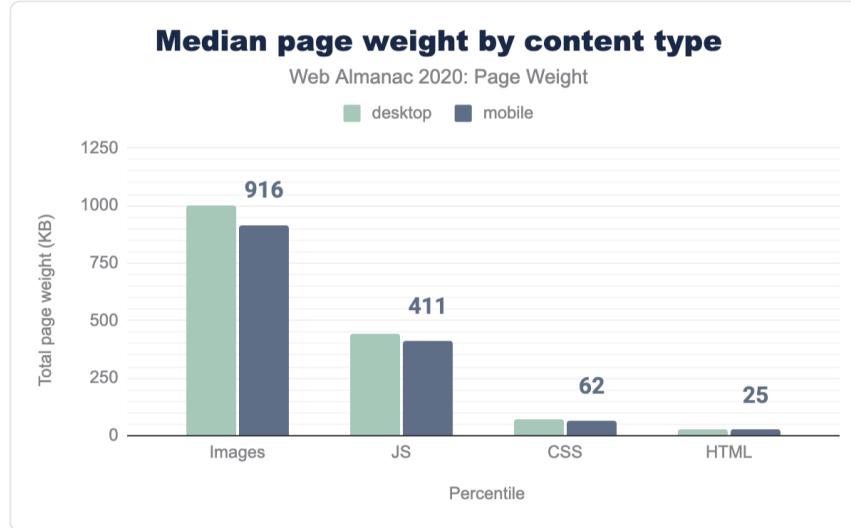


Figure 2.1. Median page weight per content type.

But nothing is free, and that's especially true for JavaScript—all that code has a cost. Let's dig in and take a closer look at how much script we use, how we use it, and what the fallout is.

How much JavaScript do we use?

We mentioned that the `script` tag is the 6th most used HTML element. Let's dig in a bit deeper to see just how much JavaScript that actually amounts to.

The median site (the 50th percentile) sends 444 KB of JavaScript when loaded on a desktop device, and slightly fewer (411 KB) to a mobile device.

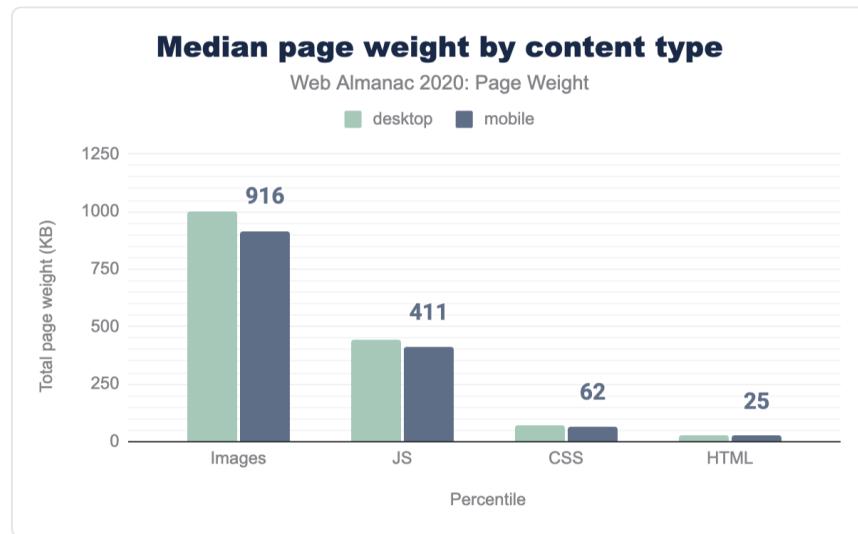


Figure 2.2. Distribution of the amount of JavaScript kilobytes loaded per page.

It's a bit disappointing that there isn't a bigger gap here. While it's dangerous to make too many assumptions about network or processing power based on whether the device in use is a phone or a desktop (or somewhere in between), it's worth noting that HTTP Archive mobile tests are done by emulating a Moto G4 and a 3G network. In other words, if there was any work being done to adapt to less-than-ideal circumstances by passing down less code, these tests should be showing it.

The trend also seems to be in favor of using more JavaScript, not less. Comparing to last year's results, at the median we see a 13.4% increase in JavaScript as tested on a desktop device, and a 14.4% increase in the amount of JavaScript sent to a mobile device.

Client	2019	2020	Change
Desktop	391	444	13.4%
Mobile	359	411	14.4%

Figure 2.3. Year-over-year change in the median number of JavaScript kilobytes per page.

At least some of this weight seems to be unnecessary. If we look at a breakdown of how much of that JavaScript is unused on any given page load, we see that the median page is shipping 152 KB of unused JavaScript. That number jumps to 334 KB at the 75th percentile and 567 KB at the 90th percentile.

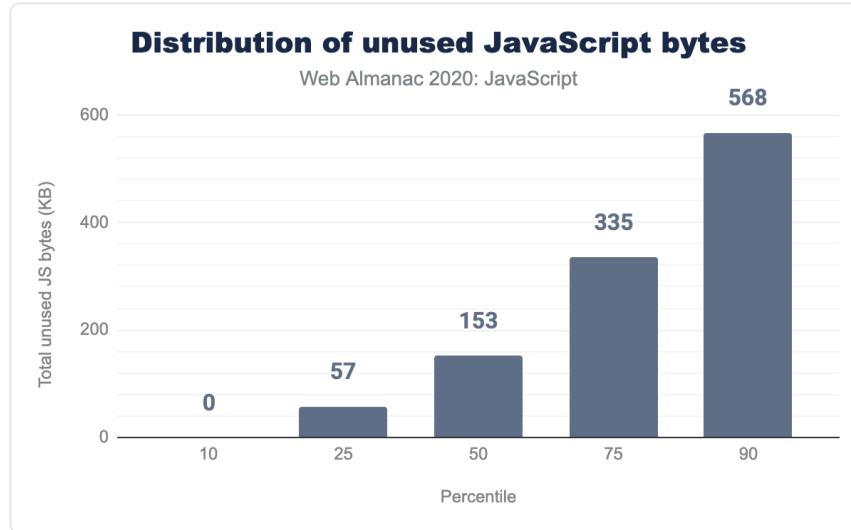


Figure 2.4. Distribution of the amount of wasted JavaScript bytes per mobile page.

As raw numbers, those may or may not jump out at you depending on how much of a performance nut you are, but when you look at it as a percentage of the total JavaScript used on each page, it becomes a bit easier to see just how much waste we're sending.

37.22%

Figure 2.5. Percent of the median mobile page's JavaScript bytes that are unused.

That 153 KB equates to ~37% of the total script size that we send down to mobile devices. There's definitely some room for improvement here.

Request count

Another way of looking at how much JavaScript we use is to explore how many JavaScript requests are made on each page. While reducing the number of requests was paramount to maintaining good performance with HTTP/1.1, with HTTP/2 the opposite is the case: breaking JavaScript down into smaller, individual files is typically better for performance.

Distribution of JavaScript requests per page (2020)

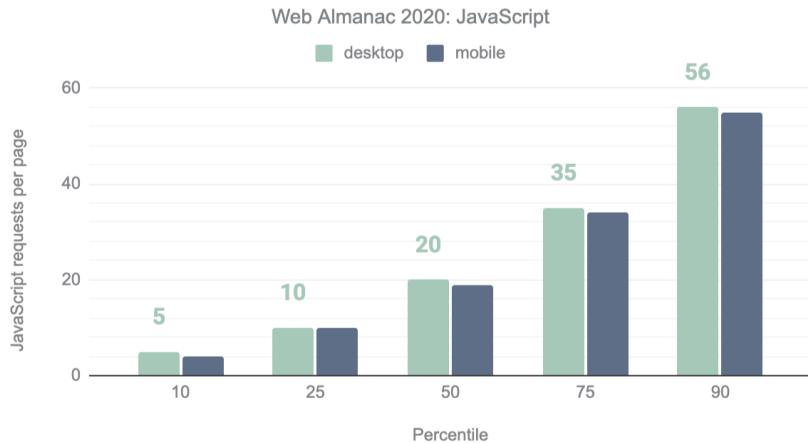


Figure 2.6. Distribution of JavaScript requests per page.

At the median, pages make 20 JavaScript requests. That's only a minor increase over last year, when the median page made 19 JavaScript requests.

Distribution of JavaScript requests per page (2019)

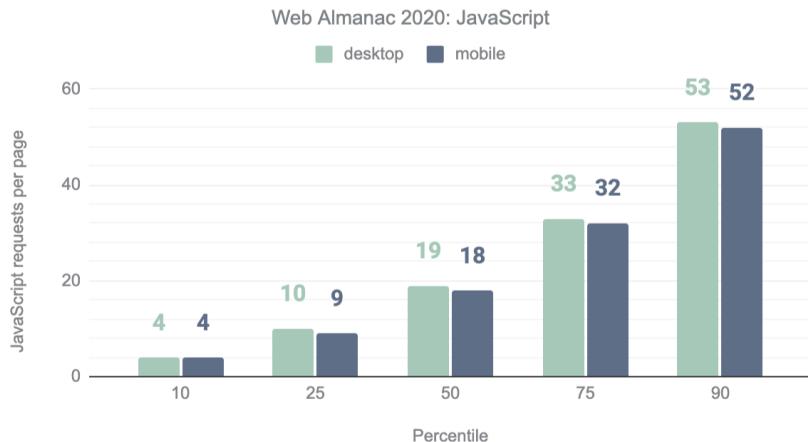


Figure 2.7. Distribution of JavaScript requests per page in 2019.

Where does it come from?

One trend that likely contributes to the increase in JavaScript used on our pages is the seemingly ever-increasing amount of third-party scripts that get added to pages to help with everything from client-side A/B testing and analytics, to serving ads and handling personalization.

Let's drill into that a bit to see just how much third-party script we're serving up.

Right up until the median, sites serve roughly the same number of first-party scripts as they do third-party scripts. At the median, 9 scripts per page are first-party, compared to 10 per page from third-parties. From there, the gap widens a bit: the more scripts a site serves in the total, the more likely it is that the majority of those scripts are from third-party sources.

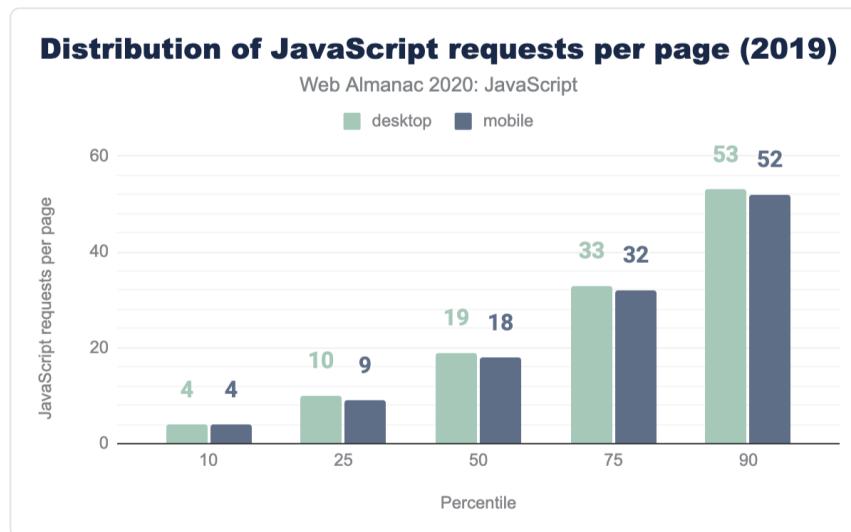


Figure 2.8. Distribution of the number of JavaScript requests by host for desktop.

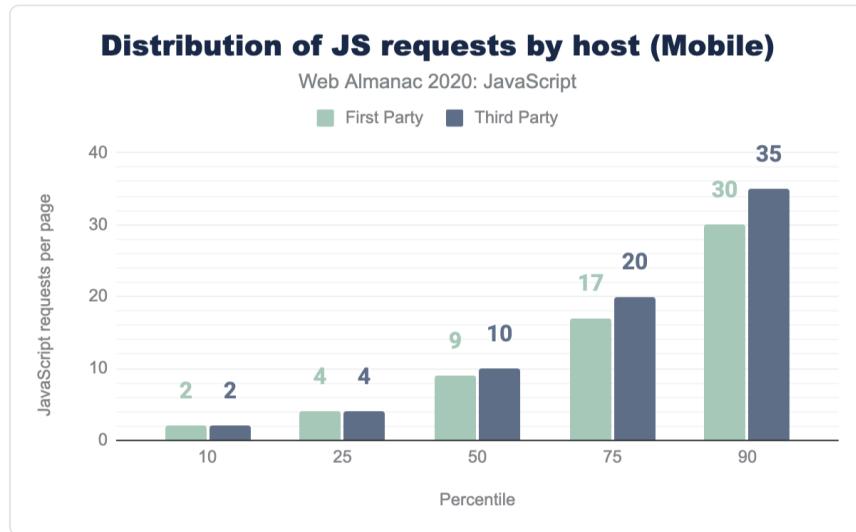


Figure 2.9. Distribution of the number of JavaScript requests by host for mobile.

While the amount of JavaScript requests are similar at the median, the actual size of those scripts is weighted (pun intended) a bit more heavily toward third-party sources. The median site sends 267 KB of JavaScript from third-parties to desktop devices ,compared to 147 KB from first-parties. The situation is very similar on mobile, where the median site ships 255 KB of third-party scripts compared to 134 KB of first-party scripts.

Distribution of JS bytes by host (Desktop)

Web Almanac 2020: JavaScript

First Party Third Party

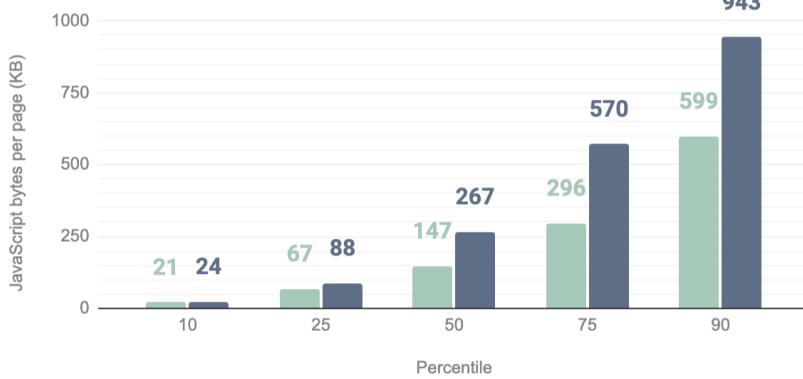


Figure 2.10. Distribution of the number of JavaScript bytes by host for desktop.

Distribution of JS bytes by host (Mobile)

Web Almanac 2020: JavaScript

First Party Third Party

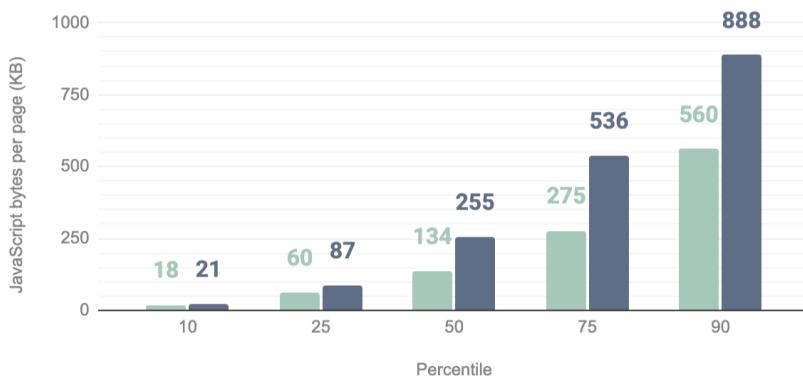


Figure 2.11. Distribution of the number of JavaScript bytes by host for mobile.

How do we load our JavaScript?

The way we load JavaScript has a significant impact on the overall experience.

By default, JavaScript is *parser-blocking*. In other words, when the browser discovers a `script` element, it must pause parsing of the HTML until the script has been downloaded, parsed, and executed. It's a significant bottleneck and a common contributor to pages that are slow to render.

We can start to offset some of the cost of loading JavaScript by loading scripts either asynchronously, which only halts the HTML parser during the parse and execution phases and not during the download phase, or deferred, which doesn't halt the HTML parser at all. Both attributes are only available on external scripts—inline scripts cannot have them applied.

On mobile, external scripts comprise 59.0% of all script elements found.

As an aside, when we talk about how much JavaScript is loaded on a page earlier, that total doesn't account for the size of these inline scripts—because they're part of the HTML document, they're counted against the markup size. This means we load even more script than the numbers show.

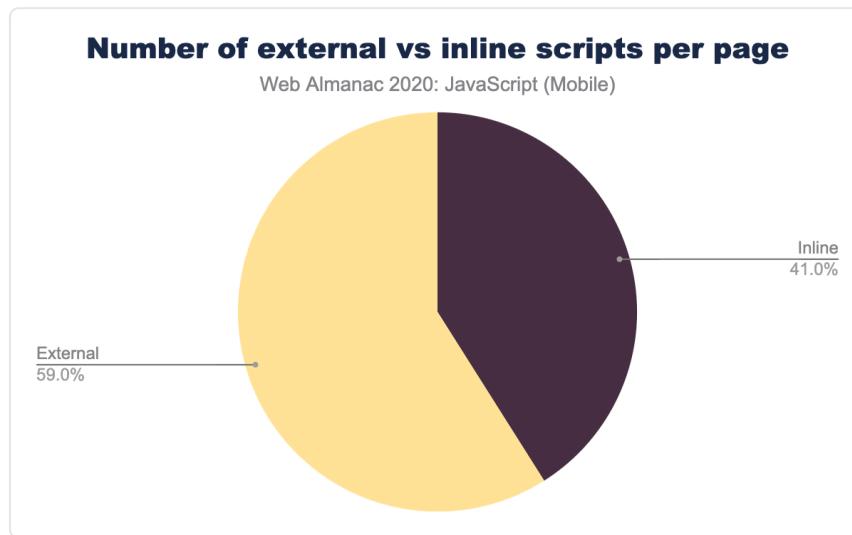


Figure 2.12. Distribution of the number of external and inline scripts per mobile page.

Of those external scripts, only 12.2% of them are loaded with the `async` attribute and 6.0% of them are loaded with the `defer` attribute.

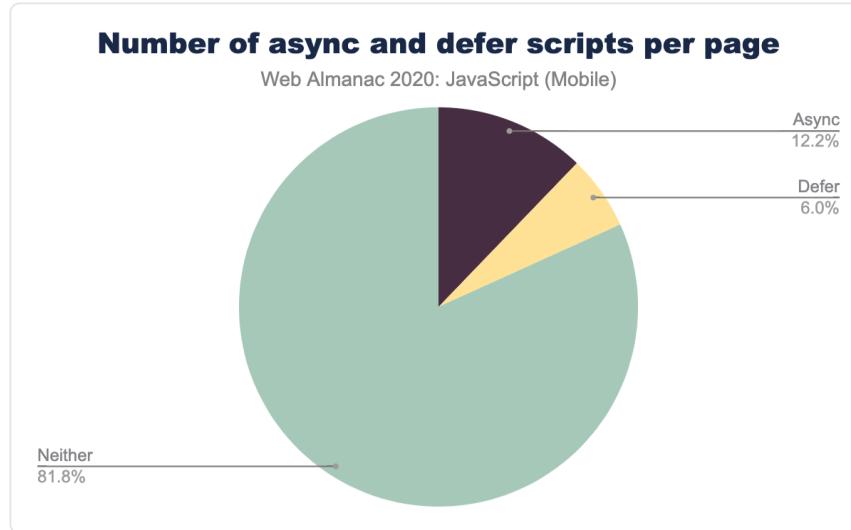


Figure 2.13. Distribution of the number of `async` and `defer` scripts per mobile page.

Considering that `defer` provides us with the best loading performance (by ensuring downloading the script happens in parallel to other work, and execution waits until after the page can be displayed), we would hope to see that percentage a bit higher. In fact, as it is that 6.0% is a bit misleading.

Back when supporting IE8 and IE9 was more common, it was relatively common to use *both* the `async` and `defer` attributes. With both attributes in place, any browser supporting both will use `async`. IE8 and IE9, which don't support `async` will fall back to `defer`.

Nowadays, the pattern is unnecessary for the vast majority of sites and any script loaded with the pattern in place will interrupt the HTML parser when it needs to be executed, instead of deferring until the page has loaded. The pattern is still used surprisingly often, with 11.4% of mobile pages serving at least one script with that pattern in place.

Resource hints

Another tool we have at our disposal for offsetting some of the network costs of loading JavaScript are resource hints, specifically, `prefetch` and `preload`.

The `prefetch` hint lets developers signify that a resource will be used on the next page navigation, therefore the browser should try to download it when the browser is idle.

The `preload` hint signifies that a resource will be used on the current page and that the

browser should download it right away at a higher priority.

Overall, we see 16.7% of mobile pages using at least one of the two resource hints to load JavaScript more proactively.

Of those, nearly all of the usage is coming from `preload`. While 16.6% of mobile pages use at least one `preload` hint to load JavaScript, only 0.4% of mobile pages use at least one `prefetch` hint.

There's a risk, particularly with `preload`, of using too many hints and reducing their effectiveness, so it's worth looking at the pages that do use these hints to see how many they're using.

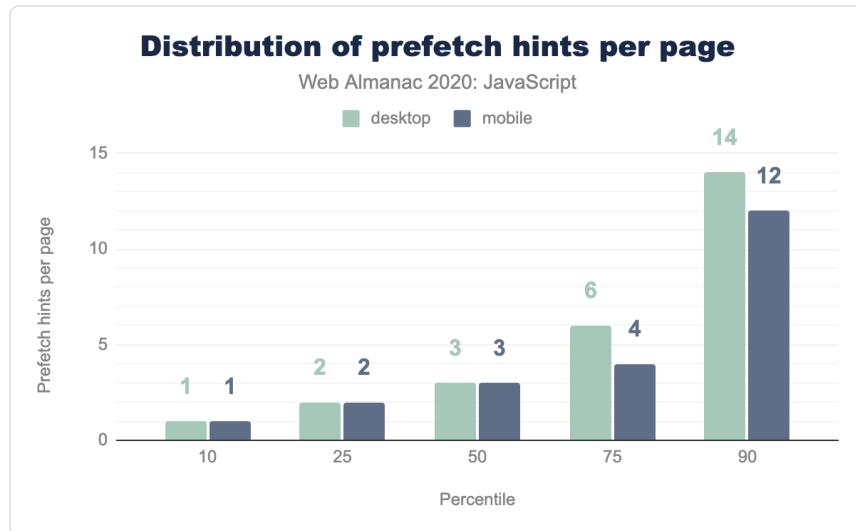


Figure 2.14. Distribution of the number `prefetch` hints per page with any `prefetch` hints.

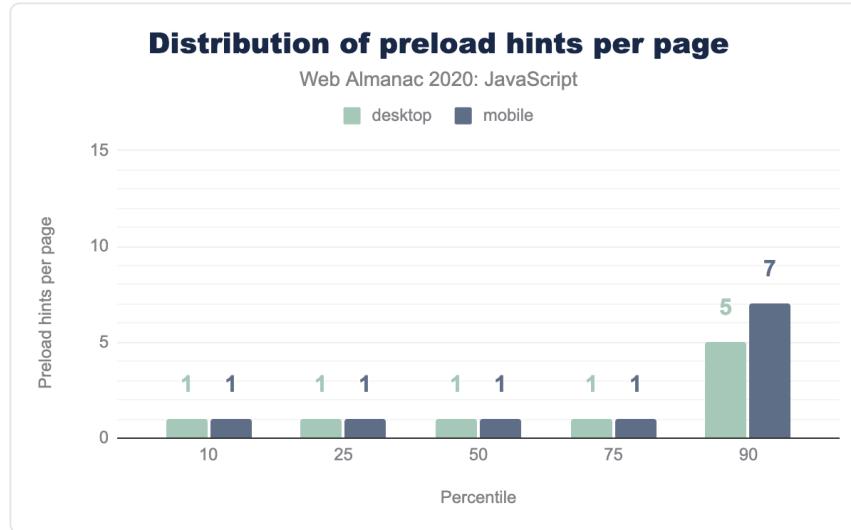


Figure 2.15. Distribution of the number `preload` hints per page with any `preload` hints.

At the median, pages that use a `prefetch` hint to load JavaScript use three, while pages that use a `preload` hint only use one. The long tail gets a bit more interesting, with 12 `prefetch` hints used at the 90th percentile and 7 `preload` hints used on the 90th as well. For more detail on resource hints, check out this year's Resource Hints chapter.

How do we serve JavaScript?

As with any text-based resource on the web, we can save a significant number of bytes through minimization and compression. Neither of these are new optimizations—they've been around for quite awhile—so we should expect to see them applied in more cases than not.

One of the audits in Lighthouse checks for unminified JavaScript, and provides a score (0.00 being the worst, 1.00 being the best) based on the findings.

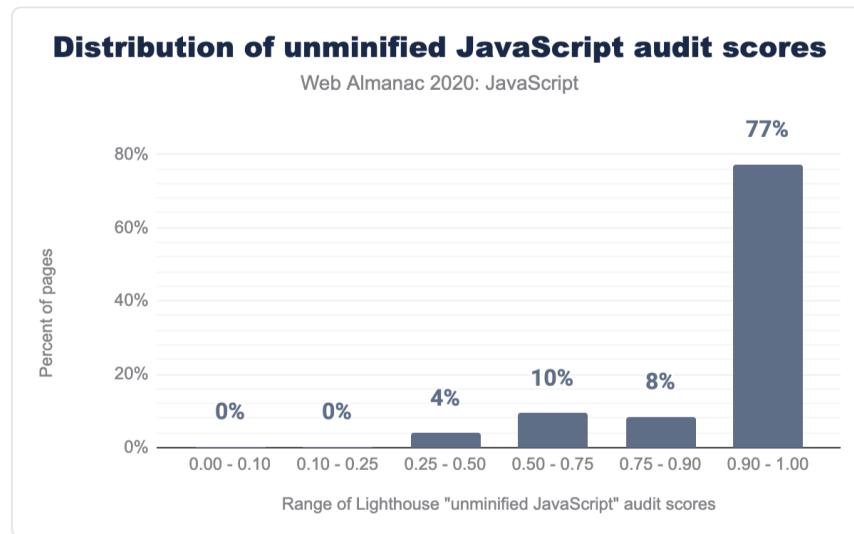


Figure 2.16. Distribution of unminified JavaScript Lighthouse audit scores per mobile page.

The chart above shows that most pages tested (77%) get a score of 0.90 or above, meaning that few unminified scripts are found.

Overall, only 4.5% of the JavaScript requests recorded are unminified.

Interestingly, while we've picked on third-party requests a bit, this is one area where third-party scripts are doing better than first-party scripts. 82% of the average mobile page's unminified JavaScript bytes come from first-party code.

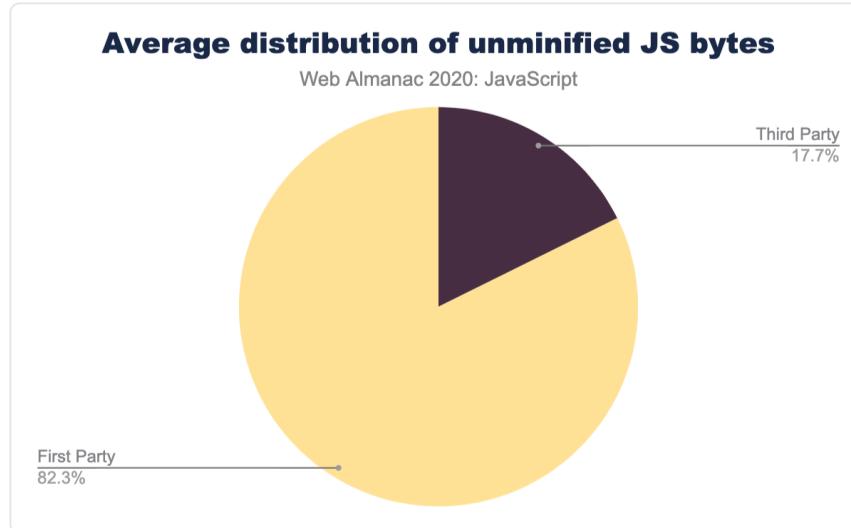


Figure 2.17. Average distribution of unminified JavaScript bytes by host.

Compression

Minification is a great way to help reduce file size, but compression is even more effective and, therefore, more important—it provides the bulk of network savings more often than not.

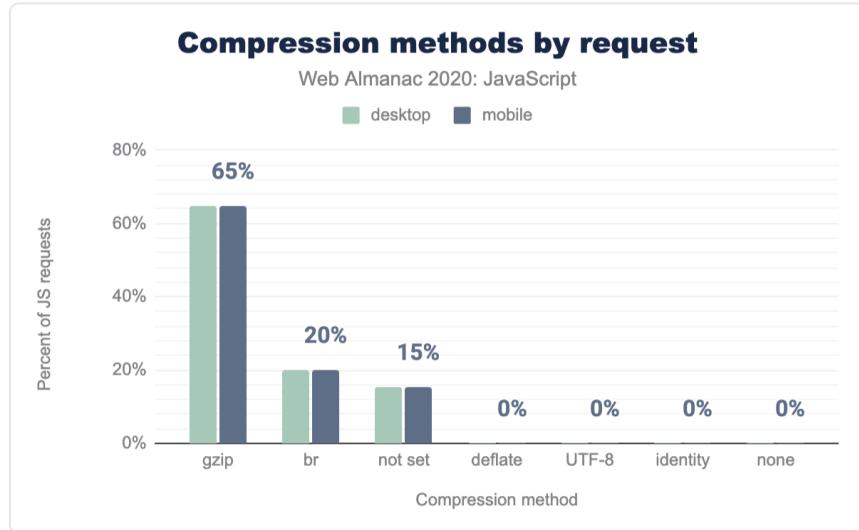


Figure 2.18. Distribution of the percent of JavaScript requests by compression method.

85% of all JavaScript requests have some level of network compression applied. Gzip makes up the majority of that, with 65% of scripts having Gzip compression applied compared to 20% for Brotli (br). While the percentage of Brotli (which is more effective than Gzip) is low compared to its browser support, it's trending in the right direction, increasing by 5 percentage points in the last year.

Once again, this appears to be an area where third-party scripts are actually doing better than first-party scripts. If we break the compression methods out by first- and third-party, we see that 24% of third-party scripts have Brotli applied, compared to only 15% of third-party scripts.

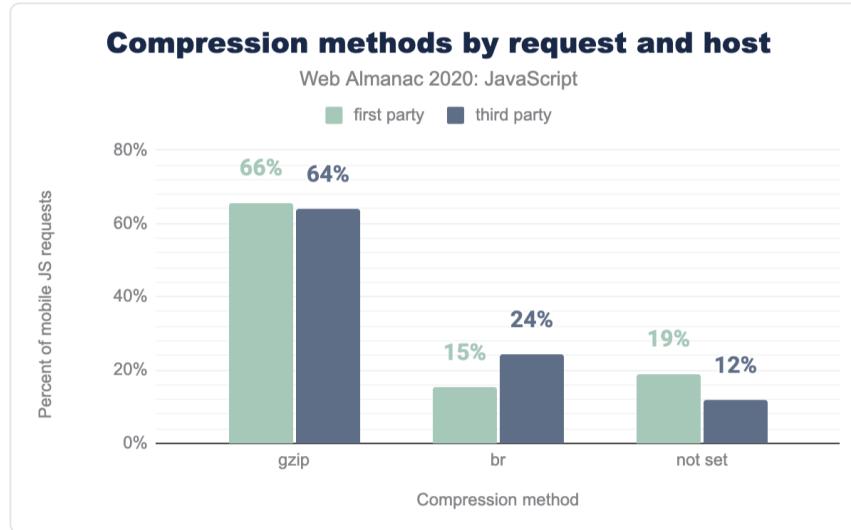


Figure 2.19. Distribution of the percent of mobile JavaScript requests by compression method and host.

Third-party scripts are also least likely to be served without any compression at all: 12% of third-party scripts have neither Gzip nor Brotli applied, compared to 19% of first-party scripts.

It's worth taking a closer look those scripts that *don't* have compression applied. Compression becomes more efficient in terms of savings the more content it has to work with. In other words, if the file is tiny, sometimes the cost of compressing the file doesn't outweigh the minuscule reduction in file size.

90.25%

Figure 2.20. Percent of uncompressed third-party JavaScript requests under 5 KB.

Thankfully, that's exactly what we see, particularly in third-party scripts where 90% of uncompressed scripts are less than 5 KB in size. On the other hand, 49% of uncompressed first-party scripts are less than 5 KB and 37% of uncompressed first-party scripts are over 10 KB. So while we do see a lot of small uncompressed first-party scripts, there are still quite a few that would benefit from some compression.

What do we use?

As we've increasingly used more JavaScript to power our sites and applications, there has also been an increasing demand for open-source libraries and frameworks to help with improving developer productivity and overall code maintainability. Sites that *don't* wield one of these tools are definitely the minority on today's web—jQuery alone is found on nearly 85% of the mobile pages tracked by HTTP Archive.

It's important that we think critically about the tools we use to build the web and what the trade-offs are, so it makes sense to look closely at what we see in use today.

Libraries

HTTP Archive uses Wappalyzer to detect technologies in use on a given page. Wappalyzer tracks both JavaScript libraries (think of these as a collection of snippets or helper functions to ease development, like jQuery) and JavaScript frameworks (these are more likely scaffolding and provide templating and structure, like React).

The popular libraries in use are largely unchanged from last year, with jQuery continuing to dominate usage and only one of the top 21 libraries falling out (*lazy.js*, replaced by *DataTables*). In fact, even the percentages of the top libraries has barely changed from last year.

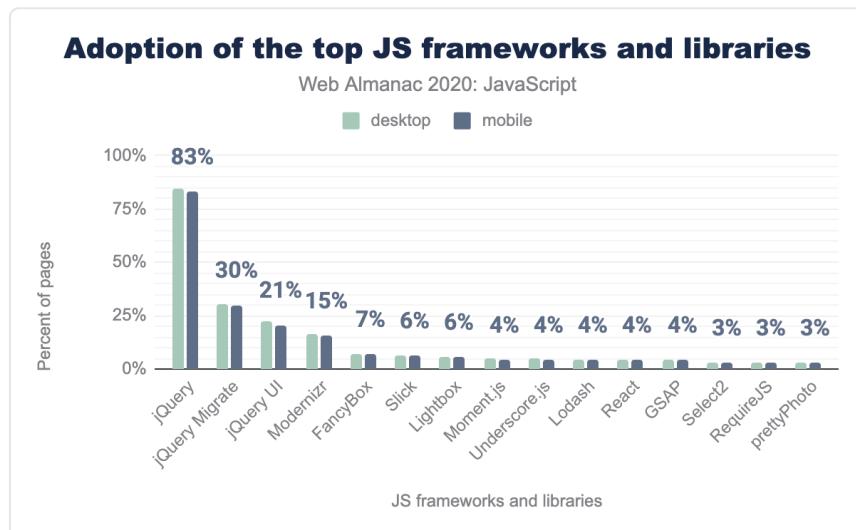


Figure 2.21. Adoption of the top JavaScript frameworks and libraries as a percent of pages.

Last year, Houssein posited a few reasons for why jQuery's dominance continues:

WordPress, which is used in more than 30% of sites, includes jQuery by default. Switching from jQuery to a newer client-side library can take time depending on how large an application is, and many sites may consist of jQuery in addition to newer client-side libraries.

Both are very sound guesses, and it seems the situation hasn't changed much on either front.

In fact, the dominance of jQuery is supported even further when you stop to consider that, of the top 10 libraries, 6 of them are either jQuery or require jQuery in order to be used: jQuery UI, jQuery Migrate, FancyBox, Lightbox and Slick.

Frameworks

When we look at the frameworks, we also don't see much of a dramatic change in terms of adoption in the main frameworks that were highlighted last year. Vue.js has seen a significant increase, and AMP grew a bit, but most of them are more or less where they were a year ago.

It's worth noting that the detection issue that was noted last year still applies, and still impacts the results here. It's possible that there *has* been a significant change in popularity for a few more of these tools, but we just don't see it with the way the data is currently collected.

What it all means

More interesting to us than the popularity of the tools themselves is the impact they have on the things we build.

First, it's worth noting that while we may think of the usage of one tool versus another, in reality, we rarely only use a single library or framework in production. Only 21% of pages analyzed report only one library or framework. Two or three frameworks are pretty common, and the long-tail gets very long, very quickly.

When we look at the common combinations that we see in production, most of them are to be expected. Knowing jQuery's dominance, it's unsurprising that most of the popular combinations include jQuery and any number of jQuery-related plugins.

Combinations	Pages	(%)
jQuery	1,312,601	20.7%
jQuery, jQuery Migrate	658,628	10.4%
jQuery, jQuery UI	289,074	4.6%
Modernizr, jQuery	155,082	2.4%
jQuery, jQuery Migrate, jQuery UI	140,466	2.2%
Modernizr, jQuery, jQuery Migrate	85,296	1.3%
FancyBox, jQuery	84,392	1.3%
Slick, jQuery	72,591	1.1%
GSAP, Lodash, React, RequireJS, Zepto	61,935	1.0%
Modernizr, jQuery, jQuery UI	61,152	1.0%
Lightbox, jQuery	60,395	1.0%
Modernizr, jQuery, jQuery Migrate, jQuery UI	53,924	0.8%
Slick, jQuery, jQuery Migrate	51,686	0.8%
Lightbox, jQuery, jQuery Migrate	50,557	0.8%
FancyBox, jQuery, jQuery UI	44,193	0.7%
Modernizr, YUI	42,489	0.7%
React, jQuery	37,753	0.6%
Moment.js, jQuery	32,793	0.5%
FancyBox, jQuery, jQuery Migrate	31,259	0.5%
MooTools, jQuery, jQuery Migrate	28,795	0.5%

Figure 2.22. The most popular combinations of libraries and frameworks on mobile pages.

We do also see a fair amount of more "modern" frameworks like React, Vue, and Angular paired with jQuery, for example as a result of migration or inclusion by third-parties.

Combination	Without jQuery	With jQuery
GSAP, Lodash, React, RequireJS, Zepto	1.0%	
React, jQuery		0.6%
React	0.4%	
React, jQuery, jQuery Migrate		0.4%
Vue.js, jQuery		0.3%
Vue.js	0.2%	
AngularJS, jQuery		0.2%
GSAP, Hammer.js, Lodash, React, RequireJS, Zepto	0.2%	
Grand Total	1.7%	1.4%

Figure 2.23. The most popular combinations of React, Angular, and Vue with and without jQuery.

More importantly, all these tools typically mean more code and more processing time.

Looking specifically at the frameworks in use, we see that the median JavaScript bytes for pages using them varies dramatically depending on what is being used.

The graph below shows the median bytes for pages where any of the top 35 most commonly detected frameworks were found, broken down by client.

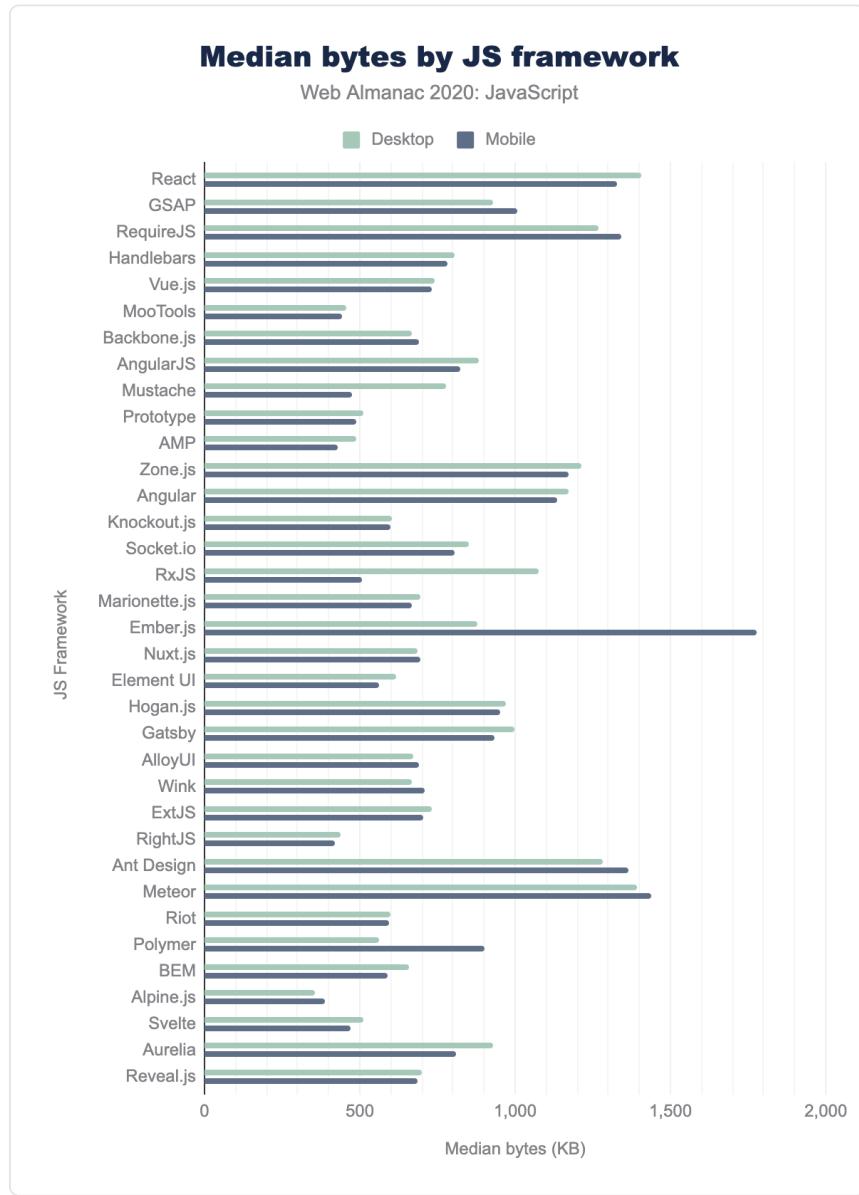


Figure 2.24. The median number of JavaScript kilobytes per page by JavaScript framework.

On one end of the spectrum are frameworks like React or Angular or Ember, which tend to ship a lot of code regardless of the client. On the other end, we see minimalist frameworks like Alpine.js and Svelte showing very promising results. Defaults are very important, and it seems that by

starting with highly performant defaults, Svelte and Alpine are both succeeding (so far... the sample size is pretty small) in creating a lighter set of pages.

We get a very similar picture when looking at main thread time for pages where these tools were detected.

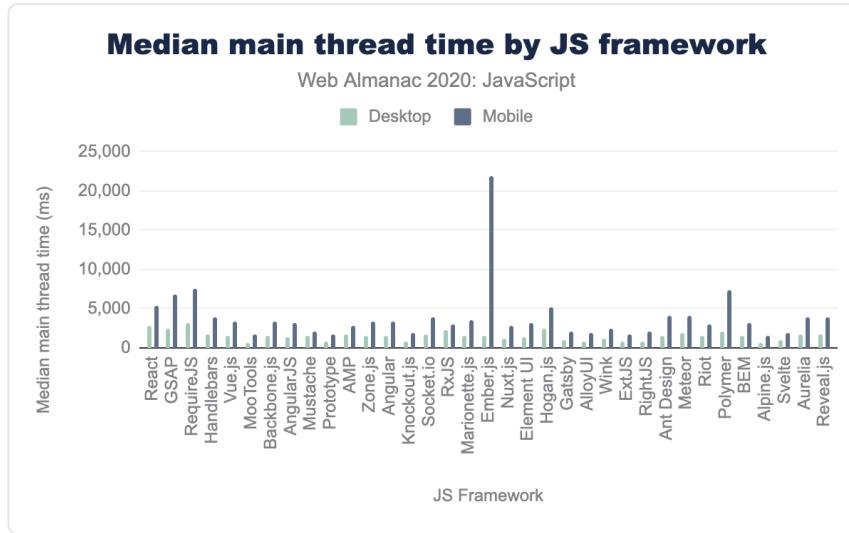


Figure 2.25. The median main thread time per page by JavaScript framework.

Ember's mobile main thread time jumps out and kind of distorts the graph with how long it takes. Pulling it out makes the picture a bit easier to understand.

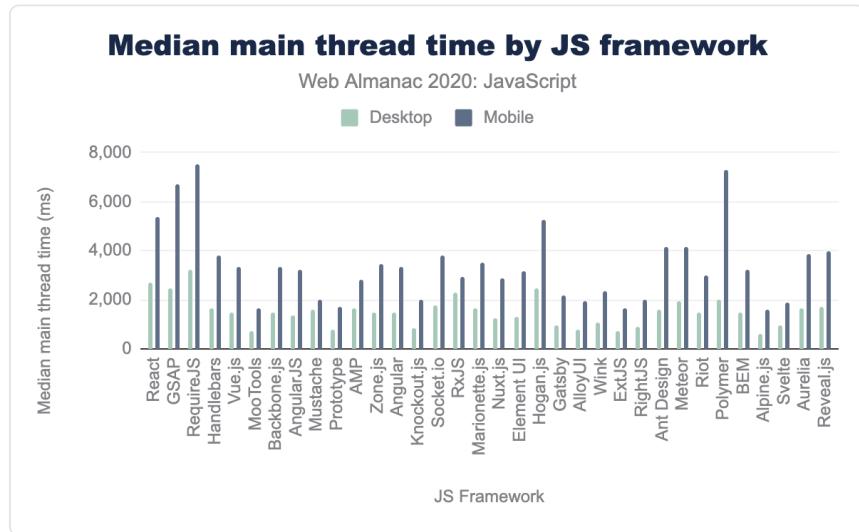


Figure 2.26. The median main thread time per page by JavaScript framework, excluding Ember.js.

Tools like React, GSAP, and RequireJS tend to spend a lot of time on the main thread of the browser, regardless of whether it's a desktop or mobile page view. The same tools that tend to lead to less code overall—tools like Alpine and Svelte—also tend to lead to lower impact on the main thread.

The gap between the experience a framework provides for desktop and mobile is also worth digging into. Mobile traffic is becoming increasingly dominant, and it's critical that our tools perform as well as possible for mobile pageviews. The bigger the gap we see between desktop and mobile performance for a framework, the bigger the red flag.

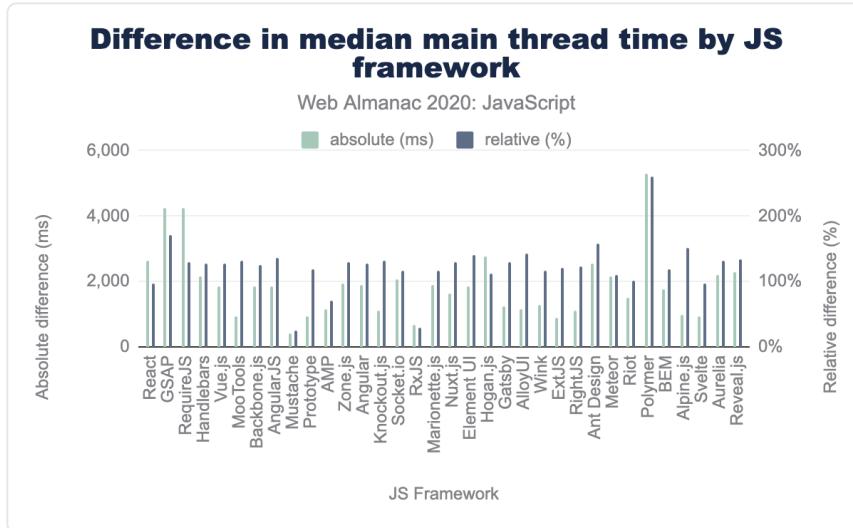


Figure 2.27. Difference between desktop and mobile median main thread time per page by JavaScript framework, excluding Ember.js.

As you would expect, there's a gap for all tools in use due to the lower processing power of the emulated Moto G4. Ember and Polymer seem to jump out as particularly egregious examples, while tools like RxJS and Mustache vary only minorly from desktop to mobile.

What's the impact?

We have a pretty good picture now of how much JavaScript we use, where it comes from, and what we use it for. While that's interesting enough on its own, the real kicker is the "so what?" What impact does all this script actually have on the experience of our pages?

The first thing we should consider is what happens with all that JavaScript once it's been downloaded. Downloading is only the first part of the JavaScript journey. The browser still has to parse all that script, compile it, and eventually execute it.

If you recall, there was only a 30 KB difference between what is shipped to a mobile device versus a desktop device. Depending on your point of view, you could be forgiven for not getting too upset about the small gap in the amount of code sent to a desktop browser versus a mobile one—after all, what's an extra 30 KB or so at the median, right?

The biggest problem comes when all of that code gets served to a low-to-middle-end device, something a bit less like the kind of devices most developers are likely to have, and a bit more

like the kind of devices you'll see from the majority of people across the world. That relatively small gap between desktop and mobile is much more dramatic when we look at it in terms of processing time.

The median desktop site spends 891 ms on the main thread of a browser working with all that JavaScript. The median mobile site, however, spends 1,897 ms—over two times the time spent on the desktop. It's even worse for the long tail of sites. At the 90th percentile, mobile sites spend a staggering 8,921 ms of main thread time dealing with JavaScript, compared to 3,838 ms for desktop sites.

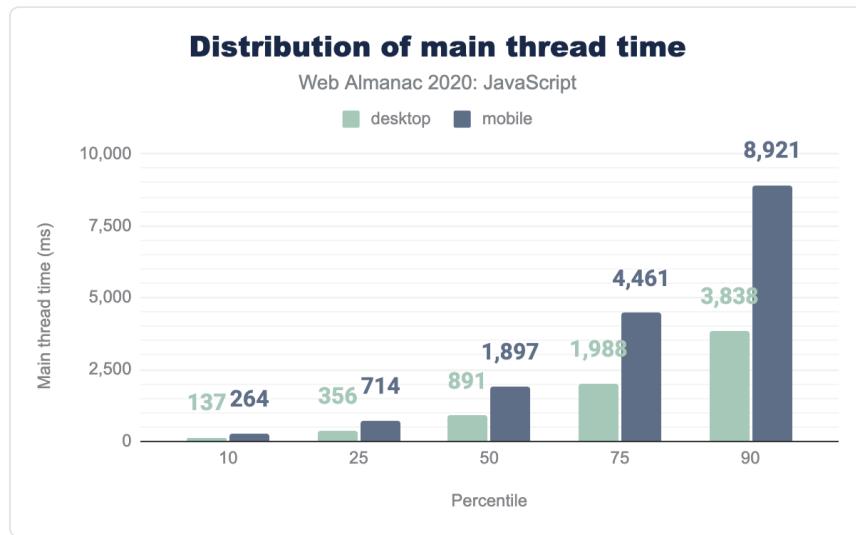


Figure 2.28. Distribution of main thread time.

Correlating JavaScript use to Lighthouse scoring

One way of looking at how this translates into impacting the user experience is to try to correlate some of the JavaScript metrics we've identified earlier with Lighthouse scores for different metrics and categories.

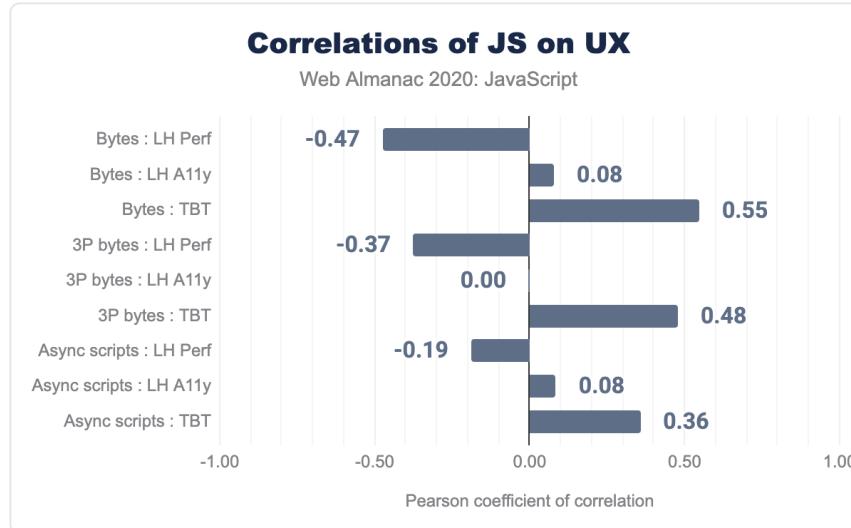


Figure 2.29. Correlations of JavaScript on various aspects of user experience.

The above chart uses the Pearson coefficient of correlation. There's a long, kinda complex definition of what that means precisely, but the gist is that we're looking for the strength of the correlation between two different numbers. If we find a coefficient of 1.00, we'd have a direct positive correlation. A correlation of 0.00 would show no connection between two numbers. Anything below 0.00 indicates a negative correlation—in other words, as one number goes up the other one decreases.

First, there doesn't seem to be much of a measurable correlation between our JavaScript metrics and the Lighthouse accessibility ("LH A11y" in the chart) score here. That stands in stark opposition to what's been found elsewhere, notably through WebAIM's annual research.

The most likely explanation for this is that Lighthouse's accessibility tests aren't as comprehensive (yet!) as what is available through other tools, like WebAIM, that have accessibility as their primary focus.

Where we do see a strong correlation is between the amount of JavaScript bytes ("Bytes") and both the overall Lighthouse performance ("LH Perf") score and Total Blocking Time ("TBT").

The correlation between JavaScript bytes and Lighthouse performance scores is -0.47. In other words, as JS bytes increase, Lighthouse performance scores decrease. The overall bytes has a stronger correlation than the amount of third-party bytes ("3P bytes"), hinting that while they certainly play a role, we can't place all the blame on third-parties.

The connection between Total Blocking Time and JavaScript bytes is even more significant

(0.55 for overall bytes, 0.48 for third-party bytes). That's not too surprising given what we know about all the work browsers have to do to get JavaScript to run in a page—more bytes means more time.

Security vulnerabilities

One other helpful audit that Lighthouse runs is to check for known security vulnerabilities in third-party libraries. It does this by detecting which libraries and frameworks are used on a given page, and what version is used of each. Then it checks Snyk's open-source vulnerability database to see what vulnerabilities have been discovered in the identified tools.

A large, bold, blue percentage value '83.50%' is displayed prominently.

Figure 2.30. Percent of mobile pages contain at least one vulnerable JavaScript library.

According to the audit, 83.5% of mobile pages use a JavaScript library or framework with at least one known security vulnerability.

This is what we call the jQuery effect. Remember how we saw that jQuery is used on a whopping 83% of pages? Several older versions of jQuery contain known vulnerabilities, which comprises the vast majority of the vulnerabilities this audit checks.

Of the roughly 5 million or so mobile pages that are tested against, 81% of them contain a vulnerable version of jQuery—a sizeable lead over the second most commonly found vulnerable library—jQuery UI at 15.6%.

Library	Vulnerable pages
jQuery	80.86%
jQuery UI	15.61%
Bootstrap	13.19%
Lodash	4.90%
Moment.js	2.61%
Handlebars	1.38%
AngularJS	1.26%
Mustache	0.77%
Dojo	0.58%
jQuery Mobile	0.53%

Figure 2.31. Top 10 libraries contributing to the highest numbers of vulnerable mobile pages according to Lighthouse.

In other words, if we can get folks to migrate away from those outdated, vulnerable versions of jQuery, we would see the number of sites with known vulnerabilities plummet (at least, until we start finding some in the newer frameworks).

The bulk of the vulnerabilities found fall into the "medium" severity category.

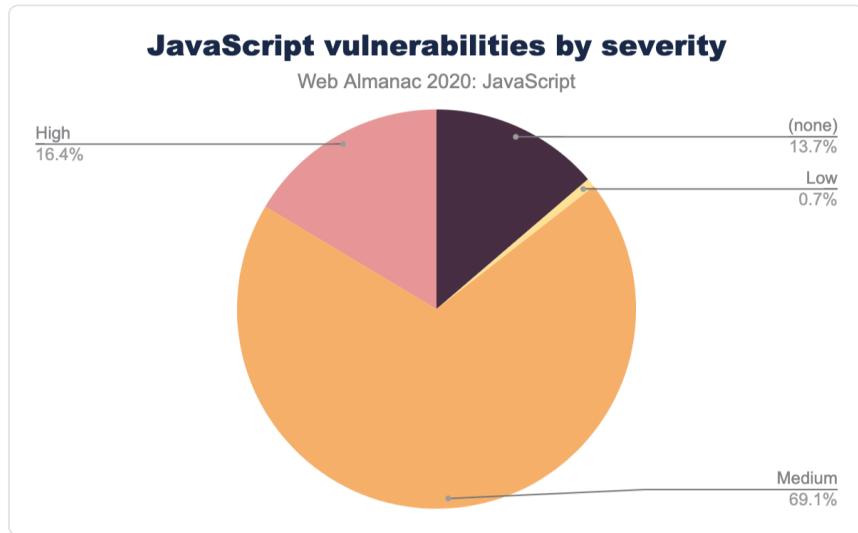


Figure 2.32. Distribution of the percent of mobile pages having JavaScript vulnerabilities by severity.

Conclusion

JavaScript is steadily rising in popularity, and there's a lot that's positive about that. It's incredible to consider what we're able to accomplish on today's web thanks to JavaScript that, even a few years ago, would have been unimaginable.

But it's clear we've also got to tread carefully. The amount of JavaScript consistently rises each year (if the stock market were that predictable, we'd all be incredibly wealthy), and that comes with trade-offs. More JavaScript is connected to an increase in processing time which negatively impacts key metrics like Total Blocking Time. And, if we let those libraries languish without keeping them updated, they carry the risk of exposing users through known security vulnerabilities.

Carefully weighing the cost of the scripts we add to our pages and being willing to place a critical eye on our tools and ask more of them are our best bets for ensuring that we build a web that is accessible, performant, and safe.

Author



Tim Kadlec

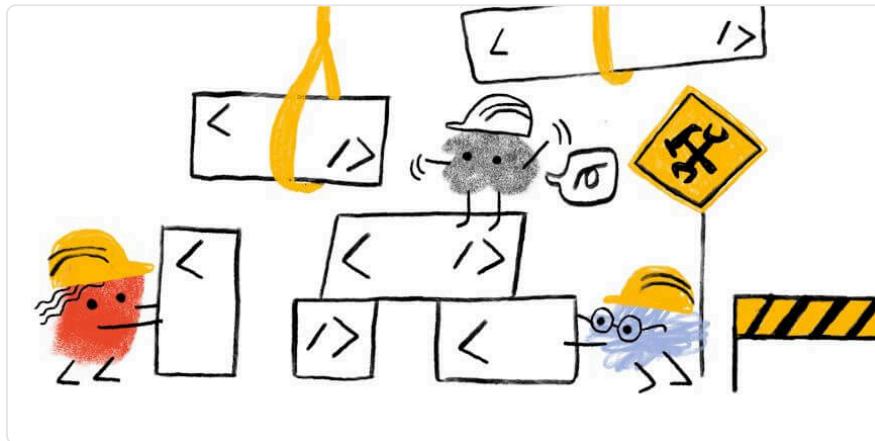
Twitter: @tkadlec | GitHub: tkadlec

Tim is a web performance consultant and trainer focused on building a web everyone can use. He is the author of *High Performance Images* (O'Reilly, 2016) and *Implementing Responsive Design* (New Riders, 2012). He writes about all things web at timkadlec.com¹⁵. You can find him sharing his thoughts in a briefer format on Twitter at @tkadlec.

15. <https://timkadlec.com/>

Part I Chapter 3

Markup



Written by Jens Oliver Meiert, Catalin Rosu, and Ian Devlin

Reviewed by Simon Pieters, Manuel Matuzovic, and Brian Kardell

Analyzed by Tony McCreathe

Introduction

The web is built on HTML. Without HTML there are no web pages, no web sites, no web apps. Nothing. There may be plain-text documents, perhaps, or XML trees, in some parallel universe that enjoyed that particular kind of challenge. In this universe, HTML is the foundation of the user-facing web. There are many standards that the web is resting on, but HTML is certainly one of the most important ones.

How do we use HTML, then, how great of a foundation do we have? In the introductory section of the 2019 Markup chapter, author Brian Kardell suggested that for a long time, we haven't really known. There were some smaller samples. For example, there was Ian Hickson's research (one of modern HTML's parents) among a few others, but until last year we lacked major insight into how we as developers, as authors, make use of HTML. In 2019 we had both Catalin Rosu's work (one of this chapter's co-authors) as well as the 2019 edition of the Web Almanac to give us a better view again of HTML in practice.

Last year's analysis was based on 5.8 million pages, of which 4.4 million were tested on desktop and 5.3 million on mobile. This year we analyzed 7.5 million pages, of which 5.6 million were tested on desktop and 6.3 million on mobile, using the latest data on the websites users are visiting in 2020. We do make some comparisons to last year, but just as we've tried to analyze additional metrics for new insights, we've also tried to impart our own personalities and perspectives throughout the chapter.

In this Markup chapter, we're focusing almost exclusively on HTML, rather than SVG or MathML, which are also considered markup languages. Unless otherwise noted, stats presented in this chapter refer to the set of mobile pages. Additionally, the data for all Web Almanac chapters is open and available. Take a look at the results and share your observations with the community!

General

In this section, we're covering the higher-level aspects of HTML like document types, the size of documents, as well as the use of comments and scripts. "Living HTML" is very much alive!

Doctypes

A large, bold, blue percentage value '96.82%' is displayed, representing the percentage of pages with a doctype.

Figure 3.1. Percent of pages with a doctype.

96.82% of pages declare a *doctype*. HTML documents declaring a doctype is useful for historical reasons, "to avoid triggering quirks mode in browsers" as Ian Hickson wrote in 2009. What are the most popular values?

Doctype	Pages	Percentage
HTML ("HTML5")	5,441,815	85.73%
XHTML 1.0 Transitional	382,322	6.02%
XHTML 1.0 Strict	107,351	1.69%
HTML 4.01 Transitional	54,379	0.86%
HTML 4.01 Transitional (quirky)	38,504	0.61%

Figure 3.2. The 5 most popular doctypes.

You can already tell how the numbers decrease quite a bit after XHTML 1.0, before entering the long tail with a few standard, some esoteric, and also bogus doctypes.

Two things stand out from these results:

1. Almost 10 years after the announcement of living HTML (aka "HTML5"), living HTML has clearly become the norm.
2. The web before living HTML can still be seen in the next most popular doctypes, like XHTML 1.0. XHTML. Although their documents are likely delivered as HTML with a MIME type of `text/html`, these older doctypes are not dead yet.

Document size

A page's document size refers to the amount of HTML bytes transferred over the network, including compression if enabled. At the extremes of the set of 6.3 million documents:

- 1,110 documents are empty (0 bytes).
- The average document size is 49.17 KB (in most cases compressed).
- The largest document by far weighs 61.19 MB, almost deserving its own analysis and chapter in the Web Almanac.

How is this situation in general, then? The median document weighs 24.65 KB, which comes without surprises:

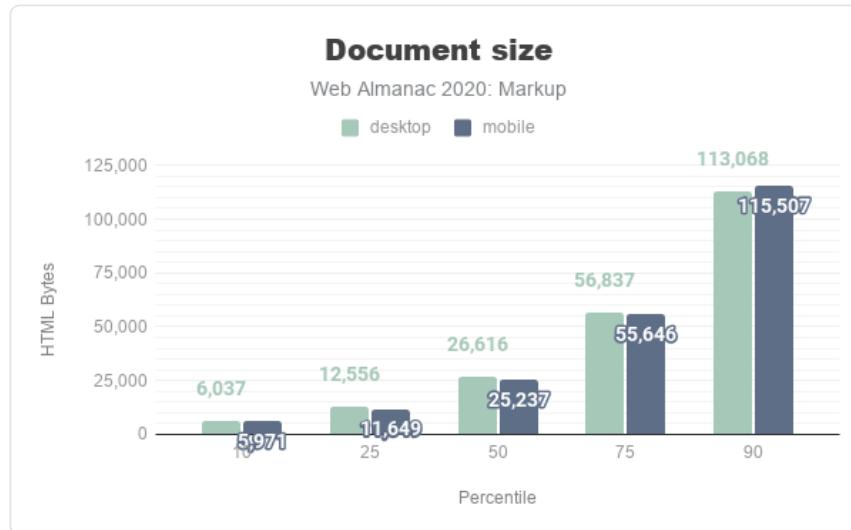


Figure 3.3. The amount of HTML bytes transferred over the network, including compression if enabled.

Document language

We identified 2,863 different values for the `lang` attribute on the `html` start tag (compare that to the 7,117 spoken languages as per Ethnologue). Almost all of them seem valid, according to the Accessibility chapter.

22.36% of all documents specify no `lang` attribute. The commonly accepted view is that they should, but beside the idea that software could eventually detect language automatically, document language can also be specified on the protocol level. This is something we didn't check.

Here are the 10 most popular (normalized) languages in our sample. It's important to note that the HTTP Archive crawls from US data centers with English language settings, so looking at the language pages will be skewed towards English. Nevertheless we present the `lang` attributes seen to give some context to the sites analyzed.

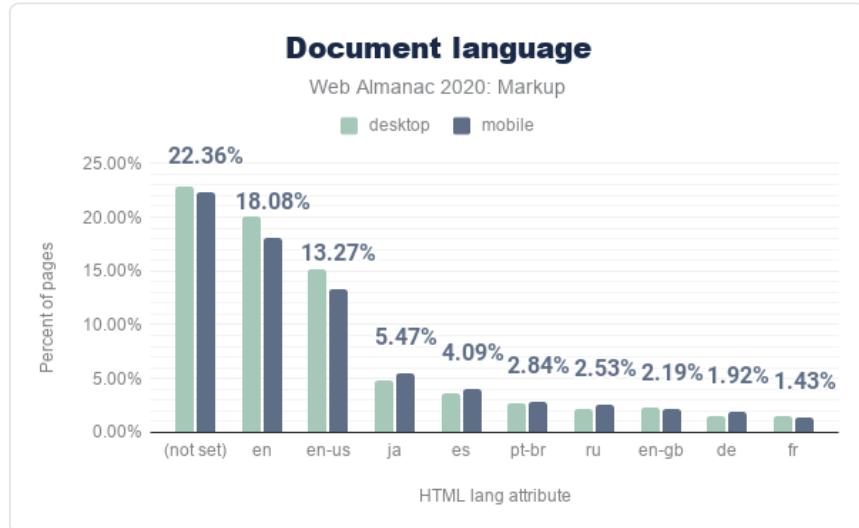


Figure 3.4. The top HTML `lang` attributes.

Comments

Adding comments to code is generally a good practice and HTML comments are there to add notes to HTML documents, without having them rendered by user agents.

```
<!-- This is a comment in HTML -->
```

Although many pages will have been stripped of comments for production, we found that index pages in the 90th percentile are using about 73 comments on mobile, respectively 79 comments on desktop, while in the 10th percentile the number of the comments is about 2. The median page uses 16 (mobile) or 17 comments (desktop).

Around 89% of pages contain at least one HTML comment, while about 46% of them contain a conditional comment.

Conditional comments

```
<!--[if IE 8]>
```

```
<p>This renders in Internet Explorer 8 only.</p>
<![endif]-->
```

The above is a non-standard HTML conditional comment. While those have proven to be helpful in the past in order to tackle browser differences, they are history for some time as Microsoft dropped conditional comments in Internet Explorer 10.

Still, on the above percentile extremes, we found that web pages are using about 6 conditional comments in the 90th percentile, and 1 comment while in the 10th percentile. Most of the pages include them for helpers such as html5shiv, selectivizr, and respond.js. While being decentish and still active pages, our conclusion is that many of them were using obsolete CMS themes.

For production, HTML comments are usually stripped by build tools. Considering all the above counts and percentages, and referring to the use of comments in general, we suppose that lots of pages are served without involving an HTML minifier.

Script use

As shown in the Top elements section below, the `script` element is the 6th most frequently used HTML element. For the purposes of this chapter, we were interested in the ways the `script` element is used across these millions of pages from the data set.

Overall, around 2% of pages contain no scripting at all, not even structured data scripts with the `type="application/ld+json"` attribute. Considering that nowadays it's pretty common for a page to include at least one script for an analytics solution, this seems noteworthy.

At the opposite end of the spectrum, the numbers show that about 97% of pages contain at least one script, either inline or external.

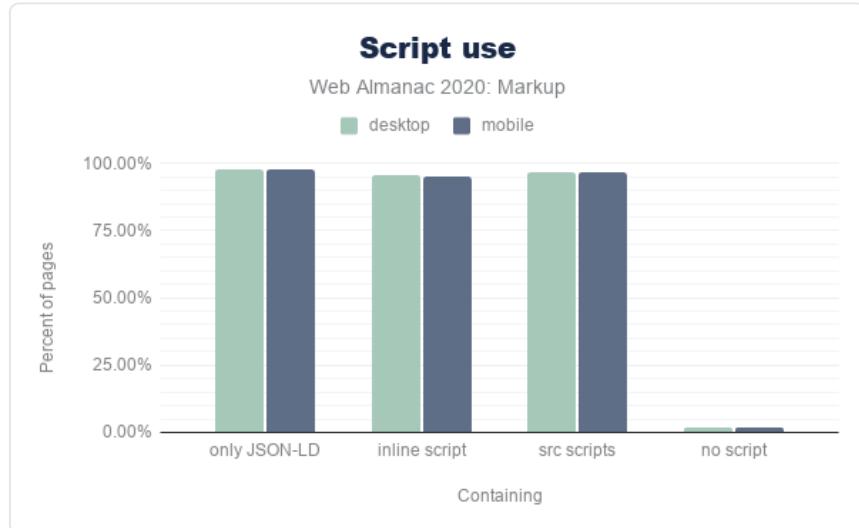


Figure 3.5. Usage of the `script` element.

When scripting is unsupported or turned off in the browser, the `noscript` element helps to add an HTML section within a page. Considering the above script numbers, we were curious about the `noscript` element as well.

Following the analysis, we found that about 49% of pages are using a `noscript` element. At the same time, about 16% of `noscript` elements were containing an `iframe` with a `src` value referring to "googletagmanager.com".

This seems to confirm the theory that the total number of `noscript` elements in the wild may be affected by common scripts like Google Tag Manager which enforce users to add a `noscript` snippet after the `body` start tag on a page.

Script types

What `type` attribute values are used with `script` elements?

- `text/javascript`: 60.03%
- `application/ld+json`: 1.68%
- `application/json`: 0.41%
- `text/template`: 0.41%
- `text/html` 0.27%

When it comes to loading JavaScript module scripts using `type="module"`, we found that

0.13% of `script` elements currently specify this attribute-value combination. `nomodule` is used by 0.95% of all tested pages. (Note that one metric relates to elements, the other to pages.)

36.38% of all scripts have no values set whatsoever.

Elements

In this section, the focus is on elements: what elements are used, how frequently, which elements are likely to appear on a given page, and how the situation is with respect to custom, obsolete, and proprietary elements. Is *divitis* still a thing? Yes.

Element diversity

Let's have a look at how diverse use of HTML actually is: Do authors use many different elements, or are we looking at a landscape that makes use of relatively few elements?

The median web page, it turns out, uses 30 different elements, 587 times:

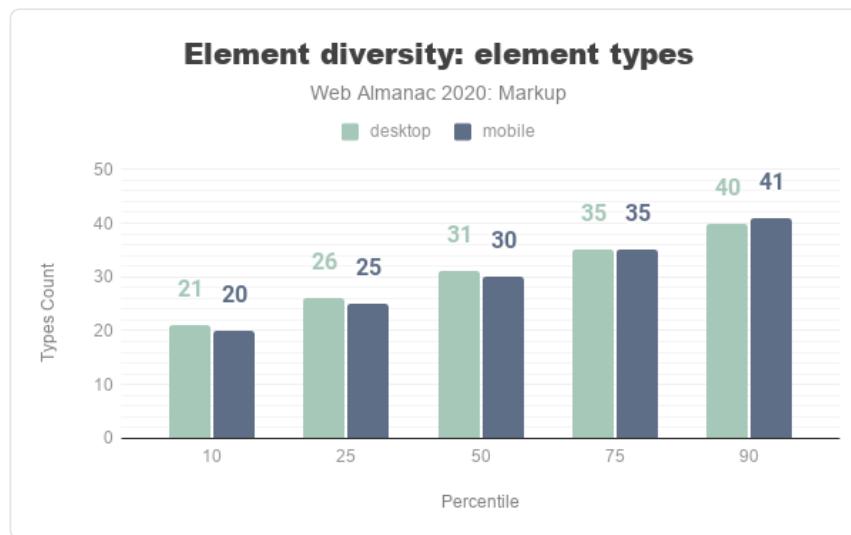


Figure 3.6. Distribution of the number of element types per page.

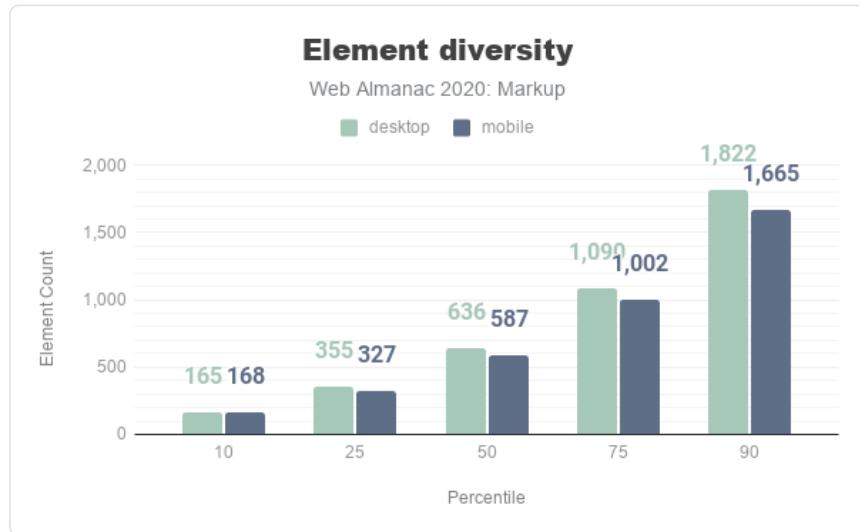


Figure 3.7. Distribution of the total number elements per page.

Given that living HTML currently has 112 elements, the 90th percentile not using more than 41 elements may suggest that HTML is not nearly being exhausted by most documents. Yet it's hard to interpret what this really means for HTML and our use of it, as the semantic wealth that HTML offers doesn't mean that every document would need all of it: HTML elements should be used per purpose (semantics), not per availability.

How are these elements distributed?

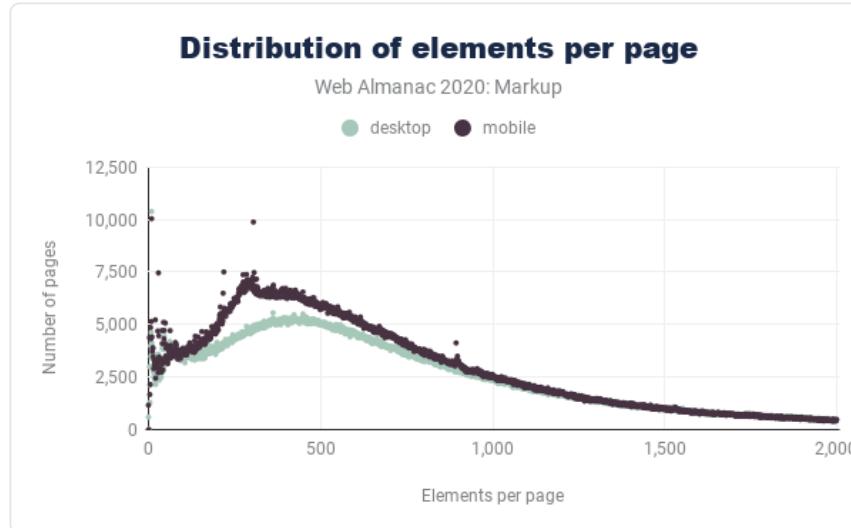


Figure 3.8. Distribution of the total number of elements per page.

Not that much changed compared to 2019!

Top elements

In 2019, the Markup chapter of the Web Almanac featured the most frequently used elements in reference to Ian Hickson's work in 2005. We found this useful and had a look at that data again:

2005	2019	2020
<i>title</i>	<i>div</i>	<i>div</i>
<i>a</i>	<i>a</i>	<i>a</i>
<i>img</i>	<i>span</i>	<i>span</i>
<i>meta</i>	<i>li</i>	<i>li</i>
<i>br</i>	<i>img</i>	<i>img</i>
<i>table</i>	<i>script</i>	<i>script</i>
<i>td</i>	<i>p</i>	<i>p</i>
<i>tr</i>	<i>option</i>	<i>link</i>
		<i>i</i>
		<i>option</i>

Figure 3.9. The most popular elements in 2005, 2019, and 2020.

Nothing changed in the Top 7, but the `option` element went a little out of favor and dropped from 8 to 10, letting both the `link` and the `i` element pass in popularity. These elements have risen in use, possibly due to an increase in use of resource hints (as with prerendering and prefetching), as well icon solutions like Font Awesome, which *de facto* misuses `i` elements for the purpose of displaying icons.

details and summary

Another thing we were curious about was the use of the `details` and `summary` elements, especially since 2020 brought broad support. Are they being used? Are they attractive for—even popular—among authors? As it turns out, only 0.39% of all tested pages are using them, although it's hard to gauge whether they were all used the correct way in exactly the situations when you need them, "popular" is the wrong word.

Here's a simple example showing the use of a `summary` in a `details` element:

```
<details>
```

```
<summary>Status: Operational</summary>
<p>Velocity: 12m/s</p>
<p>Direction: North</p>
</details>
```

A while ago, Steve Faulkner pointed out how these two elements were used inadequately in the wild. As you can tell from the example above, for each `details` element you'd need a `summary` element that may only be used as the first child of `details`.

Accordingly, we looked at the number of `details` and `summary` elements and it seems that they do continue to be misused. The count of `summary` elements is higher on both mobile and desktop, with a ratio of 1.11 `summary` elements for every `details` element on mobile, and 1.19 on desktop, respectively:

Occurrences		
Element	Mobile (0.39%)	Desktop (0.22%)
<code>summary</code>	62,992	43,936
<code>details</code>	56,60	36,743

Figure 3.10. Adoption of the `details` and `summary` elements.

Probability of element use

Taking another look at element popularity, how likely is it to find a certain element in the DOM of a page? Surely, `html`, `head`, `body` are present on every page (even though their tags are all optional), making them common elements, but what other elements are to be found?

Element	Probability
<code>title</code>	99.34%
<code>meta</code>	99.00%
<code>div</code>	98.42%
<code>a</code>	98.32%
<code>link</code>	97.79%
<code>script</code>	97.73%
<code>img</code>	95.83%
<code>span</code>	93.98%
<code>p</code>	88.71%
<code>ul</code>	87.68%

Figure 3.11. High probabilities of finding a given element in pages of the Web Almanac 2020 sample.

Standard elements are those that are or were part of the HTML specification. Which ones are you really rarely to find? In our sample, that would bring up the following:

Element	Probability
<code>dir</code>	0.0082%
<code>rp</code>	0.0087%
<code>basefont</code>	0.0092%

Figure 3.12. Low probabilities of finding a given element in pages of the sample.

We're including these elements to give an idea what elements may have gone out of favor. But while `dir` and `basefont` were last specified in XHTML 1.0 (2000), the rare use of `rp`, which has been mentioned as early as 1998 but which is also still part of HTML, may just suggest that Ruby markup is not very popular.

Custom elements

The 2019 edition of the Web Almanac handled custom elements by discussing several non-standard elements. This year, we found it valuable to have a closer look at custom elements. How did we determine these? Roughly by looking at their definition, notably their use of a hyphen. Let's focus on the top elements, in this case elements used on $\geq 1\%$ of all URLs in the sample:

Element	Occurrences	Percentage
<code>ym-measure</code>	141,156	2.22%
<code>wix-image</code>	76,969	1.21%
<code>rs-module-wrap</code>	71,272	1.12%
<code>rs-module</code>	71,271	1.12%
<code>rs-slide</code>	70,970	1.12%
<code>rs-slides</code>	70,993	1.12%
<code>rs-sbg-px</code>	70,414	1.11%
<code>rs-sbg-wrap</code>	70,414	1.11%
<code>rs-sbg</code>	70,413	1.11%
<code>rs-progress</code>	70,651	1.11%
<code>rs-mask-wrap</code>	63,871	1.01%
<code>rs-loop-wrap</code>	63,870	1.01%
<code>rs-layer-wrap</code>	63,849	1.01%
<code>wix-iframe</code>	63,590	1%

Figure 3.13. The 14 most popular custom elements.

These elements come from three sources: Yandex Metrica (`ym-`), an analytics solution we've also seen last year; Slider Revolution (`rs-`), a WordPress slider, for which there are more elements to be found near the top of the sample; and Wix (`wix-`), a website builder.

Other groups that stand out include AMP markup with `amp-` elements like `amp-img` (11,700

cases), `amp-analytics` (10,256) and `amp-auto-ads` (7,621), as well as Angular `app-` elements like `app-root` (16,314), `app-footer` (6,745), and `app-header` (5,274).

Obsolete elements

There are more questions to ask about the use of HTML, and one may relate to obsolete elements, which are elements like `applet`, `bgsound`, `blink`, `center`, `font`, `frame`, `isindex`, `marquee`, or `spacer`.

In our mobile dataset of 6.3 million pages, around 0.9 million pages (14.01%) contain one or more of these elements. Here are the top 9, which are used more than 10,000 times:

Element	Occurrences	Pages (%)
<code>center</code>	458,402	7.22%
<code>font</code>	430,987	6.79%
<code>marquee</code>	67,781	1.07%
<code>nobr</code>	31,138	0.49%
<code>big</code>	27,578	0.43%
<code>frame</code>	19,363	0.31%
<code>frameset</code>	19,163	0.30%
<code>strike</code>	17,438	0.27%
<code>noframes</code>	15,016	0.24%

Figure 3.14. Obsolete elements with more than 10,000 uses.

Even `spacer` is still being used 1,584 times, and present on every 5,000th page. We know that Google has been using a `center` element on their homepage for 22 years now, but why are there so many imitators?

`isindex`

If you were wondering: The total number of `isindex` elements in this dataset is: one. Exactly one page used an `isindex` element. It was part of the specs until HTML 4.01 and XHTML 1.0, yet only properly specified in 2006 (aligning with how it was implemented in browsers), and

then removed in 2016.

Proprietary and made-up elements

In our set of elements we found some that were neither standard HTML (nor SVG nor MathML) elements, nor custom ones, nor obsolete ones, but somewhat proprietary ones. The top 10 that we identified are the following:

Element	Pages (%)
<i>noindex</i>	0.89%
<i>jdiv</i>	0.85%
<i>mediaelementwrapper</i>	0.49%
<i>ymaps</i>	0.26%
<i>yatag</i>	0.20%
<i>ss</i>	0.11%
<i>include</i>	0.08%
<i>olark</i>	0.07%
<i>h7</i>	0.06%
<i>limespot</i>	0.05%

Figure 3.15. Elements of questionable heritage.

The source of these elements appears to be mixed, as in some are unknown while others can be traced. The most popular one, `noindex`, is probably due to Yandex's recommendation of it to prohibit page indexing. `jdiv` was noted in last year's Web Almanac and is from JivoChat. `mediaelementwrapper` comes from the MediaElement media player. Both `ymaps` and `yatag` are also from Yandex. The `ss` element could be from ProStores, a former ecommerce product from eBay, and `olark` may be from the Olark chat software. `h7` appears to be a mistake. `limespot` is probably related to the Limespot personalization program for ecommerce. None of these elements are part of a web standard.

Headings

Headings make for a special category of elements that play an important role in sectioning and for accessibility.

Heading	Occurrences	Average per page
<i>h1</i>	10,524,810	1.66
<i>h2</i>	37,312,338	5.88
<i>h3</i>	44,135,313	6.96
<i>h4</i>	20,473,598	3.23
<i>h5</i>	8,594,500	1.36
<i>h6</i>	3,527,470	0.56

Figure 3.16. Frequency and average use of standard heading elements.

You might have expected to only see the standard `<h1>` to `<h6>` elements, but some sites actually use more levels:

Heading	Occurrences	Average per page
<i>h7</i>	30,073	0.005
<i>h8</i>	9,266	0.0015

Figure 3.17. Frequency and average use of non-standard heading elements.

The last two have never been part of HTML, of course, and should not be used.

Attributes

This section focuses on how attributes are used in documents and explores patterns in `data-*` usage. Our findings show that `class` is the queen of all attributes.

Top attributes

Similar to the section on the most popular elements, this section delves into the most popular attributes on the web. Given how important the `href` attribute is for the web itself, or the `alt` attribute in order to make information accessible, would these be most popular attributes?

Attribute	Occurrences	Percentage
<code>class</code>	2,998,695,114	34.23%
<code>href</code>	928,704,735	10.60%
<code>style</code>	523,148,251	5.97%
<code>id</code>	452,110,137	5.16%
<code>src</code>	341,604,471	3.90%
<code>type</code>	282,298,754	3.22%
<code>title</code>	231,960,356	2.65%
<code>alt</code>	172,668,703	1.97%
<code>rel</code>	171,802,460	1.96%
<code>value</code>	140,666,779	1.61%

Figure 3.18. Top 10 attributes by frequency of use.

The most popular attribute is `class`, with nearly 3 billion occurrences in our dataset and constituting 34% of all attributes in use. `class` is by far the most prevalent attribute.

The `value` attribute, which specifies the value of an `input` element, surprisingly completes the top 10. It's surprising to us because, subjectively, we didn't get the impression `value` attributes were used that frequently.

Attributes on pages

Are there attributes that we find in every document? Not quite, but almost:

Element	Pages (%)
<code>href</code>	99.21%
<code>src</code>	99.18%
<code>content</code>	98.88%
<code>name</code>	98.61%
<code>type</code>	98.55%
<code>class</code>	98.24%
<code>rel</code>	97.98%
<code>id</code>	97.46%
<code>style</code>	95.95%
<code>alt</code>	90.75%

Figure 3.19. Top 10 attributes by page.

These results raise some questions that we cannot answer. For example, `type` is used on other elements too, but why this tremendous popularity? Especially given that it's usually not needed to specify for style sheets or scripts, with CSS and JavaScript being assumed default. Or, how do we really fare with `alt`? Do those 9.25% of pages not contain any images or are they just inaccessible?

data-* attributes

Per the HTML spec, `data-*` attributes "are intended to store custom data, state, annotations, and similar, private to the page or application, for which there are no more appropriate attributes or elements." How are they used? What are the popular ones? Is there anything interesting here?

The two most popular ones stand out because they are almost twice as popular than each of the attributes that followed (with >1% use):

Attribute	Occurrences	Percentage
<code>data-src</code>	26,734,560	3.30%
<code>data-id</code>	26,596,769	3.28%
<code>data-toggle</code>	12,198,883	1.50%
<code>data-slick-index</code>	11,775,250	1.45%
<code>data-element_type</code>	11,263,176	1.39%
<code>data-type</code>	11,130,662	1.37%
<code>data-requiremodule</code>	8,303,675	1.02%
<code>data-requirecontext</code>	8,302,335	1.02%

Figure 3.20. The most popular `data-*` attributes.

Attributes like `data-type`, `data-id`, and `data-src` can have multiple generic uses although `data-src` is used a lot with lazy image loading via JavaScript (e.g., Bootstrap 4). Bootstrap again explains the presence of `data-toggle`, where it's used as a state styling hook on toggle buttons. The Slick carousel plugin is the source of `data-slick-index`, whereas `data-element_type` is part of Elementor's WordPress website builder. Both `data-requiremodule` and `data-requirecontext`, then, are part of RequireJS.

Interestingly, the use of native lazy loading on images is similar to that of `data-src`. 3.86% of pages use the `` attribute. This appears to be growing very fast, as back in February, this number was about 0.8%. It's possible that these are being used together for a cross-browser solution.

Miscellaneous

We've covered the use of HTML in general as well as the adoption of top elements and attributes. In this section, we're reviewing some of the special cases of viewports, favicons, buttons, inputs, and links. One thing we note here is that too many links still point to "http" URLs.

viewport specifications

The viewport meta element is used to control layout on mobile browsers. While years ago, the motto was kind of "don't forget the viewport meta element" when building a web page, eventually this became a common practice and the motto changed to "make sure zooming and scaling are not disabled."

Users should be able to zoom and scale the text up to 500%. That's why audits in popular tools like Lighthouse or axe fail when `user-scalable="no"` is used within the `meta name="viewport"` element, and when the `maximum-scale` attribute value is less than 5.

We had a look at the data and in order to better understand the results, we normalized it by removing spaces, converting everything to lowercase, and sorting by comma values of the `content` attribute.

Content attribute value	Occurrences	Pages (%)
<code>initial-scale=1, width=device-width</code>	2,728,491	42.98%
<code>blank</code>	688,293	10.84%
<code>initial-scale=1, maximum-scale=1, width=device-width</code>	373,136	5.88%
<code>initial-scale=1, maximum-scale=1, user-scalable=no, width=device-width</code>	352,972	5.56%
<code>initial-scale=1, maximum-scale=1, user-scalable=0, width=device-width</code>	249,662	3.93%
<code>width=device-width</code>	231,668	3.65%

Figure 3.21. `viewport` specifications, and lack thereof.

The results show that almost half of the pages we analyzed are using the typical viewport `content` value. Still, around 10% of mobile pages are entirely missing a proper `content` value for the viewport meta element, with the rest of them using an improper combination of `maximum-scale`, `minimum-scale`, `user-scalable=no`, or `user-scalable=0`.

For a while now, the Edge mobile browser allows users to zoom into a web page to at least 500%, regardless of the zoom settings defined by a web page employing the viewport meta element.

Favicons

The situation around favicons is fascinating. Favicons work with or without markup—some browsers would fall back to looking at the domain root—, accept several image formats, and then also promote several dozen sizes (some tools are reported to generate 45 of them; realfavicongenerator.net would return 37 if requested to handle every case). As of this time of writing, there is an open issue for the HTML spec to help improve the situation.

When we built our tests we didn't check for the presence of images, but only looked at the markup. That means, when you review the following, note that it's more about *how* favicons are referenced rather than whether or how often they are used.

Favicon format	Occurrences	Pages (%)
ICO	2,245,646	35.38%
PNG	1,966,530	30.98%
No favicon defined	1,643,136	25.88%
JPG	319,935	5.04%
No extension specified (no format identifiable)	37,011	0.58%
GIF	34,559	0.54%
WebP	10,605	0.17%
...		
SVG	5,328	0.08%

Figure 3.22. Common favicon formats.

There are a couple of surprises in here:

- Support for other formats is there but ICO is still the go-to format for favicons on the web.
- JPG is a relatively popular favicon format even though it may not yield the best results (or a comparatively large weight) for many favicon sizes.
- WebP is twice as popular as SVG! This might change, however, with SVG favicon support improving.

Button and input types

There has been a lot of discussion on buttons lately and how often they are misused. We looked into this to present findings on some of the native HTML buttons.

60.56%

Figure 3.23. Percent of pages with button elements.

Button types	Occurrences	Percentage
<code><button type="button"></code>	15,926,061	36.41%
<code><button> without type</code>	11,838,110	32.43%
<code><button type="submit"></code>	4,842,946	28.55%
<code><input type="submit" value="..."></code>	4,000,844	31.82%
<code><input type="button" value="..."></code>	1,087,182	4.07%
<code><input type="image" src="..."></code>	322,855	2.69%
<code><button type="reset"></code>	41,735	0.49%

Figure 3.24. Adoption of button types.

Our analysis shows that about 60% of pages contain a button element and more than half of the pages (32.43%) have at least one button that fails to specify a `type` attribute. Note that the button element has a default type of `submit`, so the default behavior of buttons on these 32% of pages is to submit the current form data. To avoid possibly unexpected behavior like this, a best practice is to specify the `type` attribute.

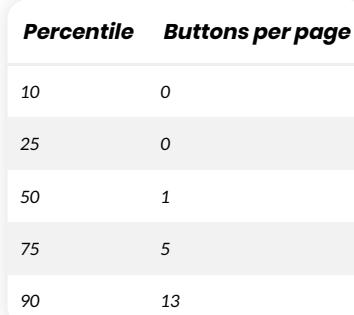


Figure 3.25. Distribution of the number of buttons per page.

Pages in the 10th and 25th percentiles contain no buttons at all, while a page in the 90th percentile contains 13 native `button` elements. In other words, 10% of pages contain 13 or more buttons.

Link targets

The anchor element, or `a` element, links web resources together. In this section, we analyze the adoption of the protocols included in these link targets.

Protocol	Occurrences	Pages (%)
<code>https</code>	5,756,444	90.69%
<code>http</code>	4,089,769	64.43%
<code>mailto</code>	1,691,220	26.64%
<code>javascript</code>	1,583,814	24.95%
<code>tel</code>	1,335,919	21.05%
<code>whatsapp</code>	34,643	0.55%
<code>viber</code>	25,951	0.41%
<code>skype</code>	22,378	0.35%
<code>sms</code>	17,304	0.27%
<code>intent</code>	12,807	0.20%

Figure 3.26. Adoption of link target protocols.

We can see how `https` and `http` are most dominant, followed by "benign" links to make writing email, making phone calls, and sending messages easier. `javascript` stands out as a link target that's still very popular even though JavaScript offers native and gracefully degrading options to work with.

Links in new windows

71.35%

Figure 3.27. Percent of pages having neither `noopener` nor `noreferrer` attributes on `target="_blank"` links.

Using `target="_blank"` has been known to be a security vulnerability for some time now. Yet 71.35% of pages contain links with `target="_blank"`, without `noopener` or `noreferrer`.

Elements	Pages
<code></code>	13.63%
<code></code>	14.14%
<code></code>	0.56%

Figure 3.28. Blank relationships.

As a rule of thumb and for usability reasons, prefer not to use `target="_blank"` in the first place.

Within the latest Safari and Firefox versions, setting `target="_blank"` on `a` elements implicitly provides the same `rel` behavior as setting `rel="noopener"`. This is already implemented in Chromium as well and will land in Chrome 88.

Conclusion

We've touched on some observations throughout the chapter, but as a reflection on the state of markup in 2020, here are some things that stood out for us:

3.97%

Figure 3.29. Percent of pages with a quirky doctype.

Fewer pages land in quirks mode. In 2016, that number was at around 7.4%. At the end of 2019, we observed 4.85%. And now, we're at about 3.97%. This trend, to paraphrase Simon Pieters in his review of this chapter, seems clear and encouraging.

Although we lack historic data to draw the full development picture, "meaningless" `div`, `span`, and `i` markup has pretty much replaced the `table` markup we've observed in the 1990s and early 2000s. While one may question whether `div` and `span` elements are always used without there being a semantically more appropriate alternative, these elements are still preferable to `table` markup, though, as during the heyday of the old web, these were seemingly used for everything but tabular data.

Elements per page and element types per page stayed roughly the same, showing no significant change in our HTML writing practice when compared to 2019. Such changes may require more time to manifest.

Proprietary product-specific elements like `g:plusone` (used on 17,607 pages in the mobile sample) and `fb:like` (11,335) have almost disappeared after still being among the most popular ones last year. However, we observe more custom elements for things like Slider Revolution, AMP, and Angular. Elements like `y-m-measure`, `jdiv`, and `ymaps` are also still prevalent. What we imagine we're seeing here is that, under the sea of slowly changing practices, HTML is very much being developed and maintained, as authors toss deprecated markup and embrace new solutions.

Now, the 2019 Web Almanac Markup chapter had 14 years of catch up to do since the last major study on the topic, so you'd think we wouldn't have much to cover in the year since. Yet what we observe with this year's data is that there's a lot of movement at the bottom and near the shore of said sea of HTML. We approach near-complete adoption of living HTML. We are quick to prune our pages of fads like Google and Facebook widgets. We're also fast in adopting and shunning frameworks, as both Angular and AMP (though a "component framework") seem to have significantly lost in popularity, likely for solutions like React and Vue.

And still, there are no signs we exhausted the options HTML gives us. The median of 30 different elements used on a given page, which is roughly a quarter of the elements HTML provides us with, suggests a rather one-sided use of HTML. That is supported by the immense popularity of elements like `div` and `span`, and no custom elements to potentially meet the

demands that these two elements may represent. Unfortunately, we couldn't validate each document in the sample; however, anecdotally and to be taken with caution, we learned that 79% of W3C-tested documents have validation errors. After everything we've seen, it looks like we're still far from mastering the craft of HTML.

That compels us to close with an appeal: Pay attention to HTML. Focus on HTML. It's important and worthwhile to invest in HTML. HTML is a document language that may not have the charm of a programming language, and yet the web is built on it. Use less HTML and learn what's really needed. Use more appropriate HTML—learn what's available and what it's there for. And validate your HTML. Anyone can write invalid HTML (just invite the next person you meet to write an HTML document and validate the output) but a professional developer can be expected to produce valid HTML. Writing correct and valid HTML is a craft to take pride in.

For the next edition of the Web Almanac's chapter, let's prepare to look closer at the craft of writing HTML and, hopefully, how we've been improving on it.

We're leaving this open to you. What are your observations? What has caught your eye? What do you think has taken a turn for the worse, and what has improved? Leave a comment to share your thoughts!

Authors



Jens Oliver Meiert

 @j9t  j9t  <https://meiert.com/en/>

Jens Oliver Meiert is a web developer and author ([CSS Optimization Basics¹⁶](https://leanpub.com/css-optimization-basics), [The Web Development Glossary¹⁷](https://leanpub.com/web-development-glossary)), who works as an engineering manager at Jimdo¹⁸. He's an expert on web development where he specializes in HTML and CSS optimization. Jens contributes to technical standards and regularly writes about his work and research, particularly on his website, [meiert.com¹⁹](https://meiert.com).

16. <https://leanpub.com/css-optimization-basics>
17. <https://leanpub.com/web-development-glossary>
18. <https://www.jimdo.com/>
19. <https://meiert.com/en/>



Catalin Rosu

Twitter: @catalinred | GitHub: catalinred | Website: <https://catalin.red/>

Catalin Rosu is a front-end developer at Caphyon²⁰ and currently works on Wattspeed²¹. He has a passion for web standards and a keen eye for great UX & UI, things he tweets²² and writes about on his website²³.



Ian Devlin

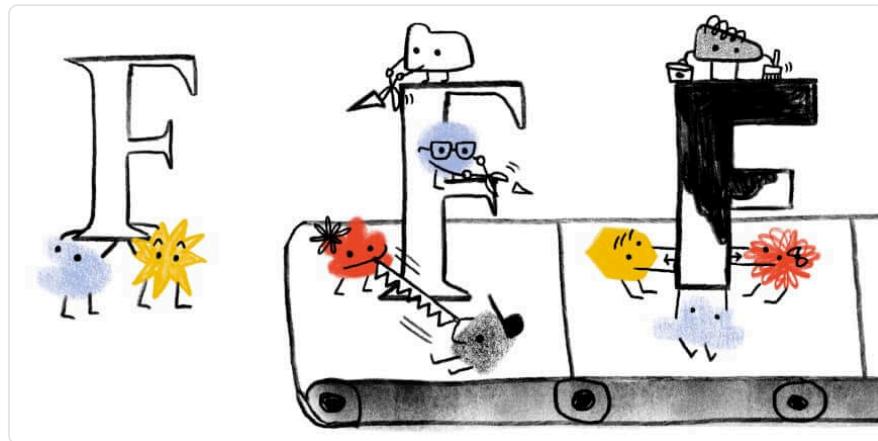
Twitter: @iandevlin | GitHub: iandevlin | Website: <https://iandevlin.com>

Ian Devlin is a web developer who advocates for good, semantic HTML, as well as accessibility. He once wrote a book about HTML5 Multimedia²⁴, and sporadically writes on his website²⁵ about the Web and other things. He currently works as a Senior Frontend Engineer at real.digital²⁶ in Germany.

20. <https://www.caphyon.com/>
21. <https://www.wattspeed.com/>
22. <https://twitter.com/catalinred>
23. <https://catalin.red/>
24. <https://www.peachpit.com/store/html5-multimedia-develop-and-design-9780321793935>
25. <https://iandevlin.com/>
26. <https://www.real-digital.de/>

Part I Chapter 4

Fonts [UNEDITED]



Written by Raph Levien and Jason Pamental

Reviewed by Roel Nieskens, Chris Lilley, Dave Crossland, rsheeter, and Mandy Michael

Analyzed by Abby Tsai

Introduction

Text is at the heart of most web sites, and typography is the art of presenting that text in a way that's visually appealing and effective. Creating good typography requires choosing the appropriate fonts. Web fonts give designers a tremendous range of fonts to choose from. As with all resources, there are performance and compatibility concerns, but done right, the benefit is well worth it. In this chapter, we'll dive into data to show how web fonts are being used, and in particular how they're optimized.

Web font technology has been fairly mature, with incremental improvements in compression and other technical improvements, but new features are arriving. Browser support for variable fonts has become quite good, and this is the feature that's seen the most growth in the previous year.

Where are web fonts being used?

Web font usage has been growing steadily over time (it was near zero as late as 2011), with 82% of web pages for desktop using web fonts, and mobile at 80%.

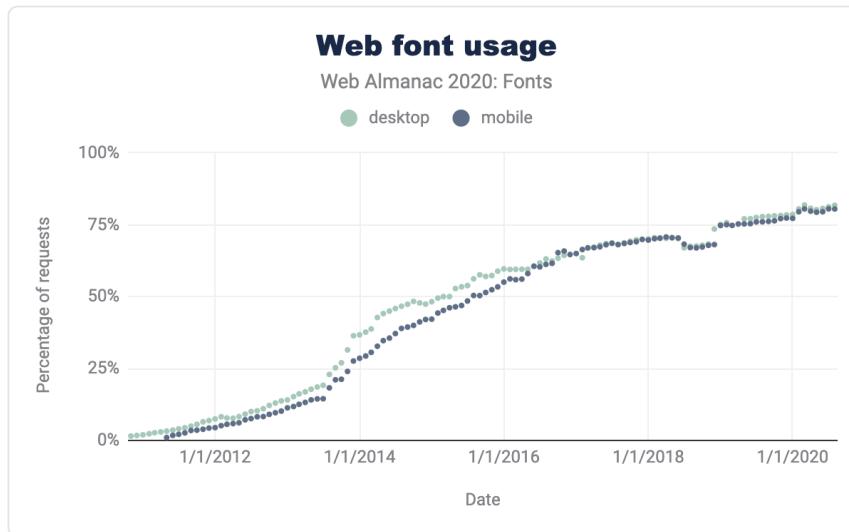


Figure 4.1. Web font usage over time.

Usage of web fonts is fairly consistent around the world, with a few outliers. The charts below are based on the median number of kilobytes of web fonts per web page, which can be an indicator of lots of fonts, large fonts, or both.

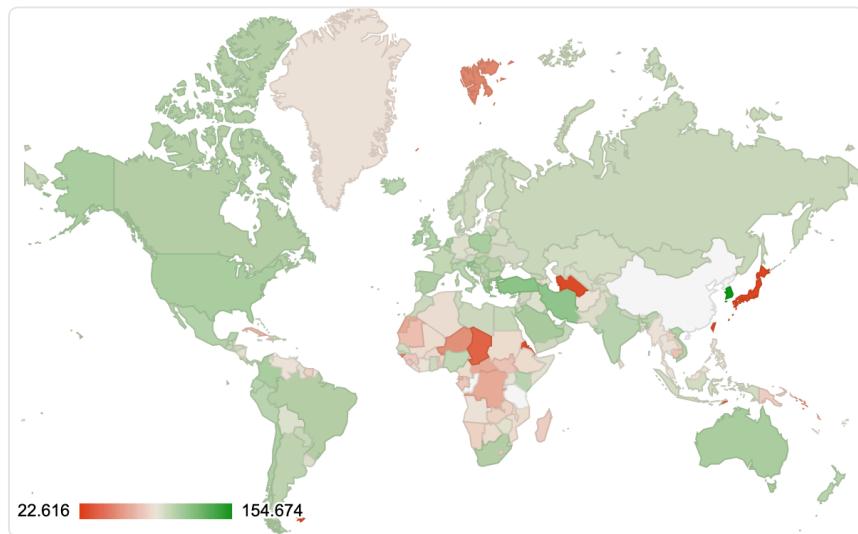


Figure 4.2. Web fonts usage by country (desktop).

The single top country is South Korea, which is not all that surprising given their consistently high internet speeds and low latency and the fact that Korean (Hangul) fonts are almost an order of magnitude larger than Latin. Web font usage in Japan and Chinese-speaking countries is considerably lower, likely because Chinese and Japanese fonts are vastly larger (the median font size can be 1000 times or more larger than the median Latin size). This means web font usage in Japan is very low, and usage in China is effectively zero. Although recent developments in progressive font enhancement (about which see more below) may make web fonts usable in both countries, within a couple of years. There have been reports that Google Fonts have not been reliably accessible in China, and that might also have been a factor holding back adoption.

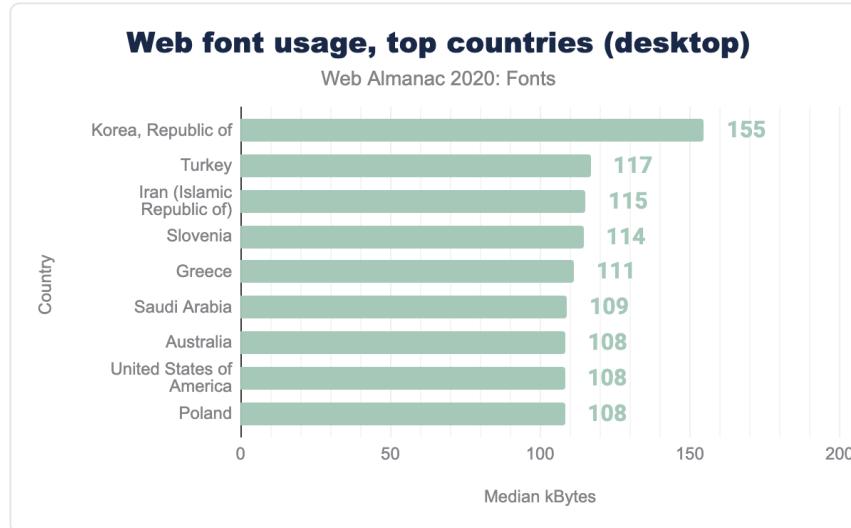


Figure 4.3. Web fonts usage, top countries (desktop).

There's a great thread on web font usage by country on the HTTP Archive discussion forum.

Serving with a service

It likely comes as no surprise that Google Fonts remains by far the most popular platform, but the percentage use has actually dropped almost 5% from 2019 to about 70%. Adobe Fonts (formerly Typekit) has dropped about 3% as well, but Bootstrap usage has grown from about 3% to over 6% (in aggregate from several providers). It's worth noting that the largest provider for Bootstrap (BootstrapCDN) also provides icon fonts from Font Awesome, so it may be that it's not Bootstrap itself but rather older versions also referencing icon font files that is behind the rise in that source data.

Another surprise in the data is the rise in fonts being served by Shopify. Growing from roughly 1.1% in 2019 to about 4% in 2020, there has clearly been a significant uptick in usage of web fonts by sites hosted on that platform. It's unclear if that is due to that service offering more fonts that they host on their CDN, if it's growth in use of their platform, or both. However, the increase in usage of both Shopify and Bootstrap represent the largest amount of growth other than Google Fonts, making it a very noticeable data point.

Not all services have the same service

It was interesting to note the differences in speed from the various free/open source and

commercial services. When looking at First Content Paint (FCP) and Last Content Paint (LCP) times, Google Fonts is roughly in the middle, but generally a bit slower than the median value. The fastest services in the dataset are Shopify and Wix (serving assets from parastorage.com), and it might be presumed they focus on a small number of highly optimized files. Google on the other hand is also serving web fonts globally of widely varying sizes (due to language), resulting in slightly slower median times.

When viewing commercial services such as Adobe (use.typekit.net) or Monotype (fast.fonts.com) it's interesting to note that on desktop they tend to be as fast or slightly faster than Google Fonts, but are noticeably slower on mobile. Conventional wisdom has generally held that the tracking scripts used by those services substantially slow them down, but that is apparently less an issue today than it has been in years past.

Local isn't always better

The use of local is controversial, as it can save bytes, but it can also yield bad results if the locally installed version of the font is outdated. As of November 2020, Google Fonts has moved to using local only for Roboto on mobile platforms, otherwise the font is always fetched over the network.

Since the data for the following charts was gathered before the switchover, Google Fonts is represented in the "both" category.

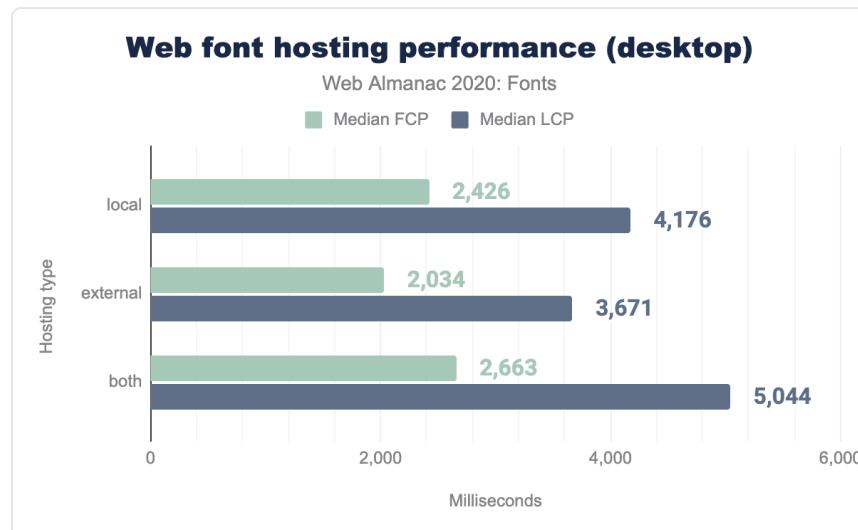


Figure 4.4. Web font hosting performance, desktop.

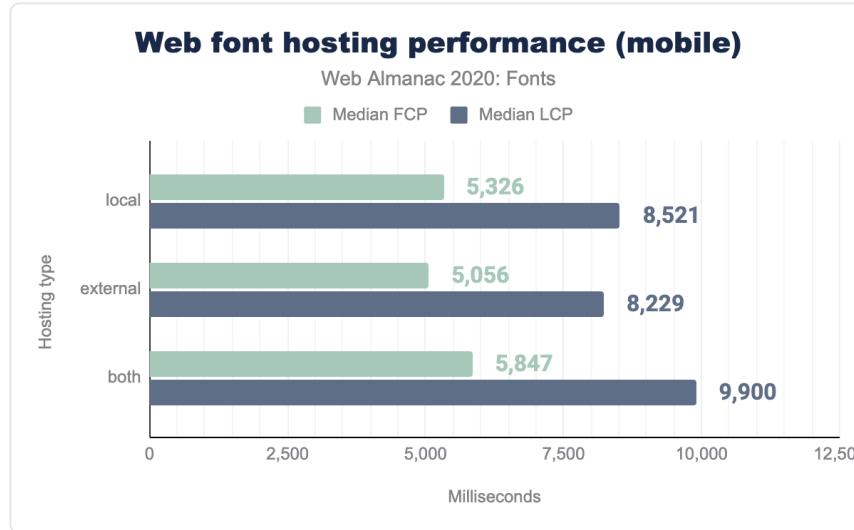


Figure 4.5. Web font hosting performance, mobile.

It wouldn't be sound to infer causality between hosting strategy from the above data, as there are other variables that may confound the relationship. But, putting that aside, we find that adding the local reference doesn't improve performance, which certainly supports the decision to remove it.

Racing to first paint

The biggest performance concern about integrating web fonts is that they may delay the time when the first readable text is displayed. Two optimization techniques can help mitigate those issues: `font-display` and resource hints.

The `font-display` setting controls what happens while waiting for the web font to load, and is generally a tradeoff between performance and visual richness. The most popular is `swap`, used on about 10% of web pages, which displays using the fallback font if the web font doesn't load quickly, then swaps in the web font when it does load. Other settings include `block`, which delays displaying text at all (minimizing the potential flashing effect), and `fallback`, which is like `swap` but gives up quickly and uses the fallback font if the font doesn't load in a moderate amount of time, and `optional`, which immediately gives up and uses the fallback font; this is used by only 1% of web pages, presumably those most concerned with performance.

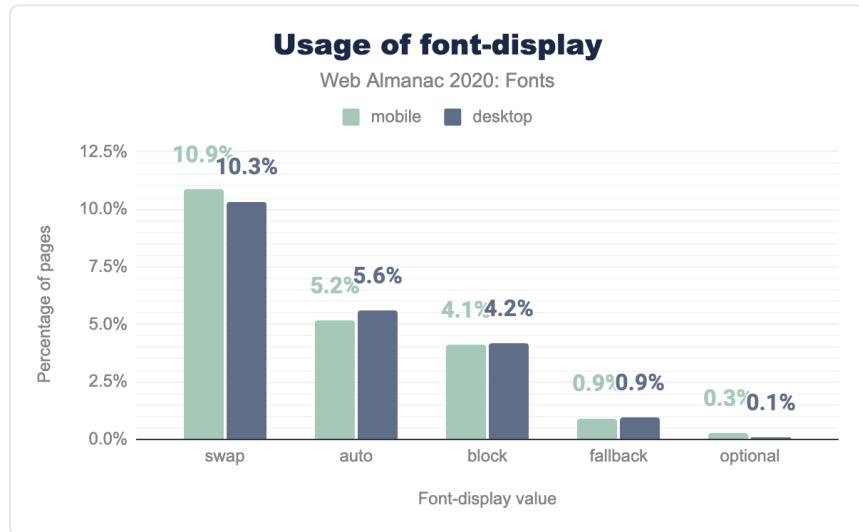


Figure 4.6. Usage of font-display.

We can analyze the effect of these settings on first content paint and last content paint. Not surprisingly, the `optional` setting has a major effect on last content paint. There is also an effect on first content paint, but that might be more correlation than causation, as all of the modes except for `block` display some text after an "extremely small block period."

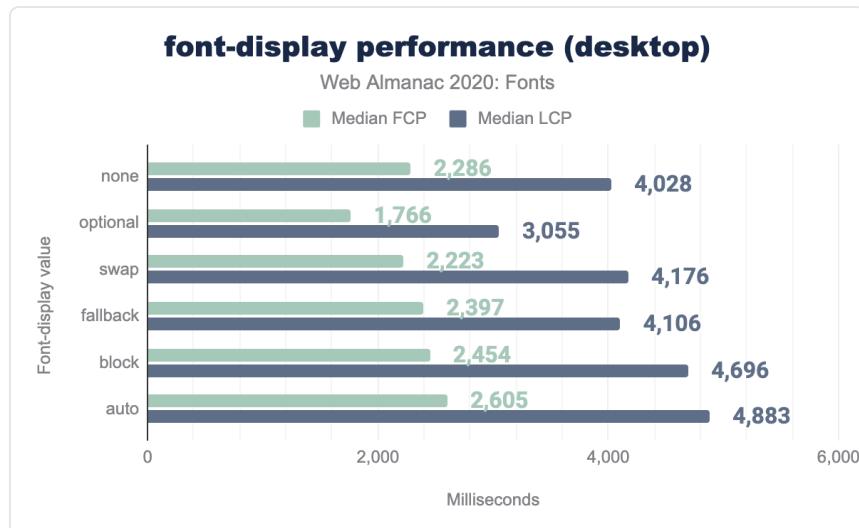


Figure 4.7. font-display, performance, desktop.

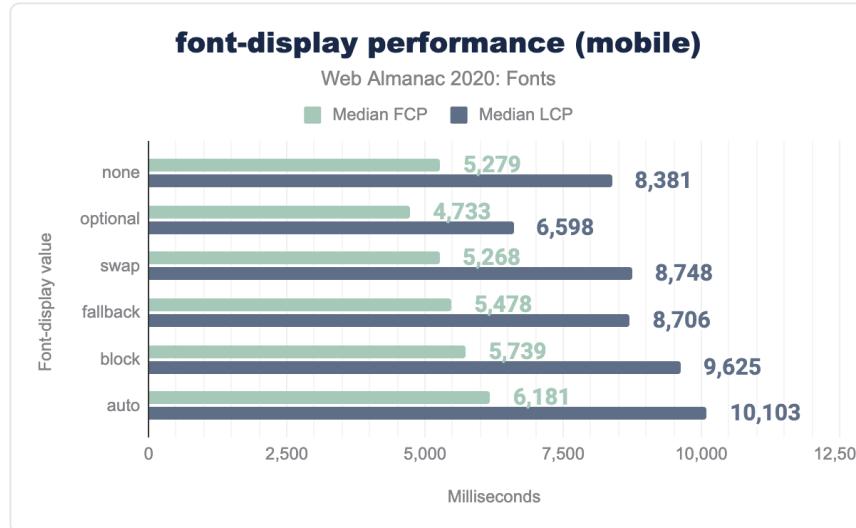


Figure 4.8. *font-display*, performance, mobile.

There are two other interesting inferences from this data. One might expect the `block` setting to have a significant impact on FCP, especially on mobile, but in practice the effect is not that large. That suggests that waiting for font assets is seldom the limiting factor for the web page performance as a whole, though it would certainly be a major factor in pages without lots of resources such as images. The `auto` setting (which is also what you get if you don't specify it) looks a lot like `block`; though it's technically up to the browser, the default is blocking in most cases.

Finally, one justification for using `fallback` is to improve Last Content Paint times compared to `swap` (which is more likely to respect the designer's visual intent), but the data do not support this case; this performance metric is no better. Perhaps this is why the setting is not popular, used by only about 1% of pages.

Google Fonts now recommends `swap` in its suggested integration code. If you're not using it now, adding it might be a way to improve performance, especially for users on slow connections.

Resource hints

While `font-display` can speed up the presentation of the page when the fonts are slow to load, resource hints can move the loading of web font assets to earlier in the cascade.

Ordinarily, fetching web fonts is a two-stage process. The first stage is loading the CSS, which

contains a reference (in `@font-face` sections) to the actual font binaries. Only then can the connection to that server begin, which further breaks down into the DNS query for the server, and actually initiating a connection (which, these days, usually involves an HTTPS cryptographic handshake).

Adding a resource hint element in the HTML starts that second connection earlier. The various resource hint settings control how far that gets before having the URL for the actual font resource. The most common (at about 32% of web pages) is `dns-prefetch`, even though in most cases there are better choices.

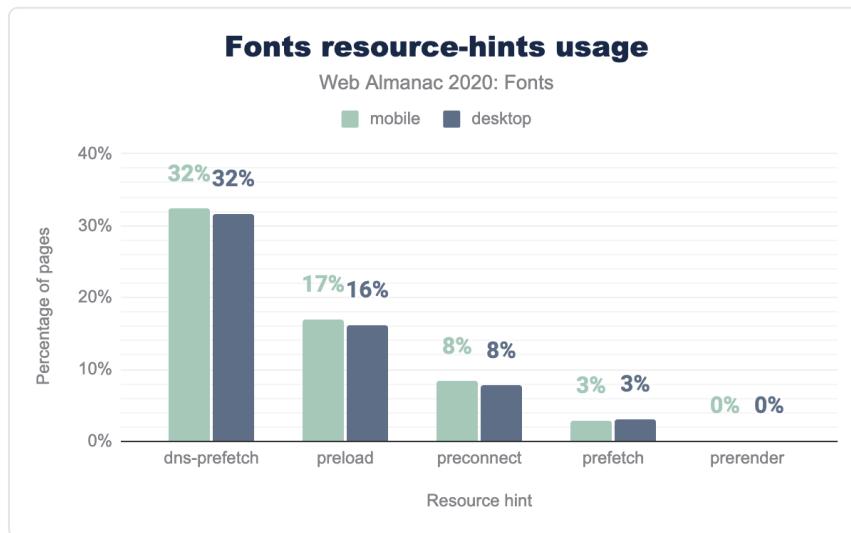


Figure 4.9. resource-hints use on fonts.

Font resource-hint performance (desktop)

Web Almanac 2020: Fonts

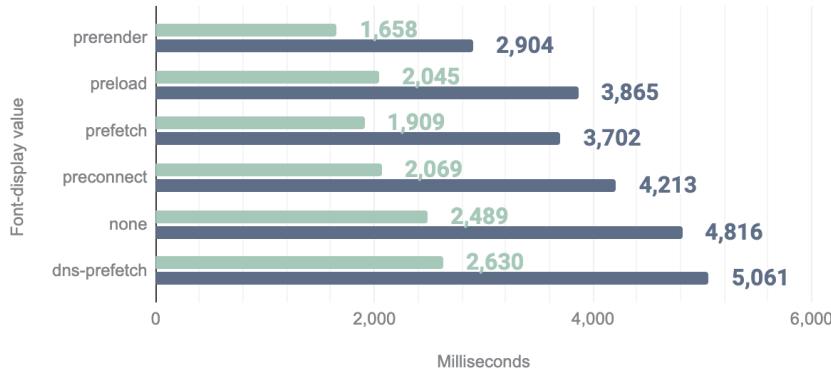
■ Median FCP ■ Median LCP


Figure 4.10. resource-hints performance, desktop.

Font resource-hint performance (mobile)

Web Almanac 2020: Fonts

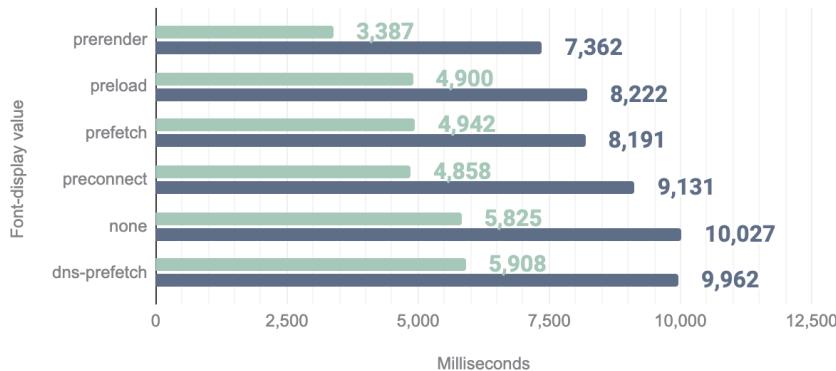
■ Median FCP ■ Median LCP


Figure 4.11. resource-hints performance, mobile.

Analysis of this data suggests that the `dns-prefetch` setting, while the most popular, doesn't improve performance much, if at all. Presumably, the DNS for popular web font servers are likely to be cached anyway. The other settings give a lot more bang for the buck, with `preconnect` being a sweet spot for ease of use, flexibility, and performance improvement. As of March 2020, Google Fonts recommends adding this line to the HTML source, immediately

before the CSS link:

```
<link rel="preconnect" href="https://fonts.gstatic.com">
```

The use of `preconnect` has grown considerably since last year, now at 8% from 2%, but there's a lot more potential performance still left on the table. Adding this line might be the single best optimization for web pages that use Google Fonts.

It might be tempting to go even farther into the pipeline, preloading or prerendering the font asset, but that potentially conflicts with other optimizations, such as fine-tuning the font for the capabilities of the rendering engine, or the `unicode-range` optimization described below. To preload a resource, you have to know *exactly* what resource to load, and the best resource for the task may depend on information not readily available at HTML authoring time.

Home on the (Unicode) range

Fonts increasingly have support for lots and lots of languages. Other fonts can have a large number of glyphs because the script (especially CJK) requires it. Either way can increase the file size. That's unfortunate if the web page is not in fact a multilingual dictionary, and only uses a fraction of the font's capabilities.

One older approach is for the HTML author to explicitly indicate a font subset. However, that requires deeper knowledge of the content, and risks a "ransom note" effect when the content uses characters supported by the font but not by the chosen subset. See the excellent essay When fonts fall by Marcin Wichary for lots more detail about how fallback works.

Static subsets, indicated by `unicode-range`, are a better approach to this problem. The font is sliced into subsets, each with a separate `@font-face` rule that indicates the Unicode coverage for that slice with a `unicode-range` descriptor. The browser then analyzes the content as part of its rendering pipeline, and downloads *only* the slices needed to render that content.

For alphabetic languages, this typically works well although it can result in poor kerning between characters in different subsets. For languages which rely on glyph shaping, such as Arabic, Urdu and many Indic languages, static subsets frequently result in broken text rendering. And for CJK, static subsets based on contiguous Unicode ranges provide almost no benefit because the characters used on a particular page are scattered almost randomly across the various subsets. Because of these issues, correct and performant use of static subsets is tricky, and requires careful analysis and implementation.

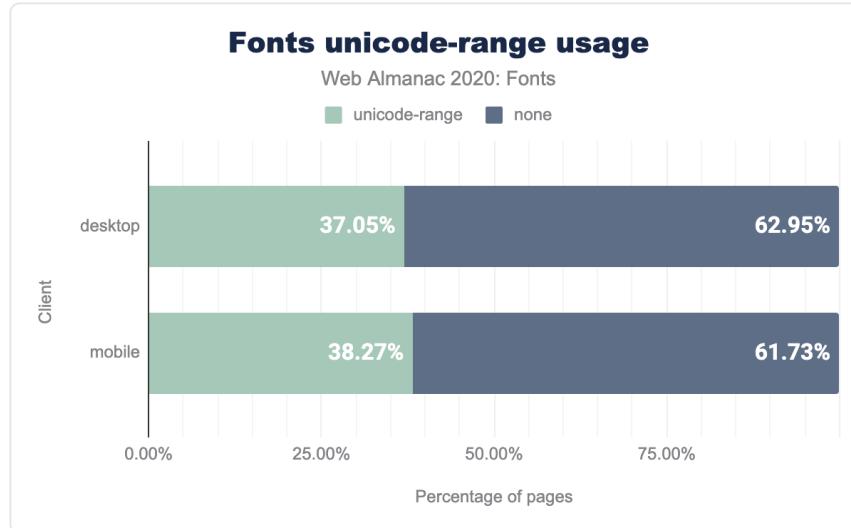


Figure 4.12. usage of `unicode-range`.

Correctly applying `unicode-range` is tricky, as there's a lot of complexity to the way text layout maps Unicode into glyphs, but Google Fonts does this automatically and transparently. It's only likely to be a win for fonts with large glyph counts. In any case, current usage is 37% on desktop and 38% on mobile.

Formats and MIME types

WOFF2 is the best compression format, and is now supported by effectively all browsers except for versions 11 and earlier of Internet Explorer. It's *almost* possible to serve web fonts using an `@font-face` rule with a WOFF2 source only. This format makes up about 75% of all fonts served.

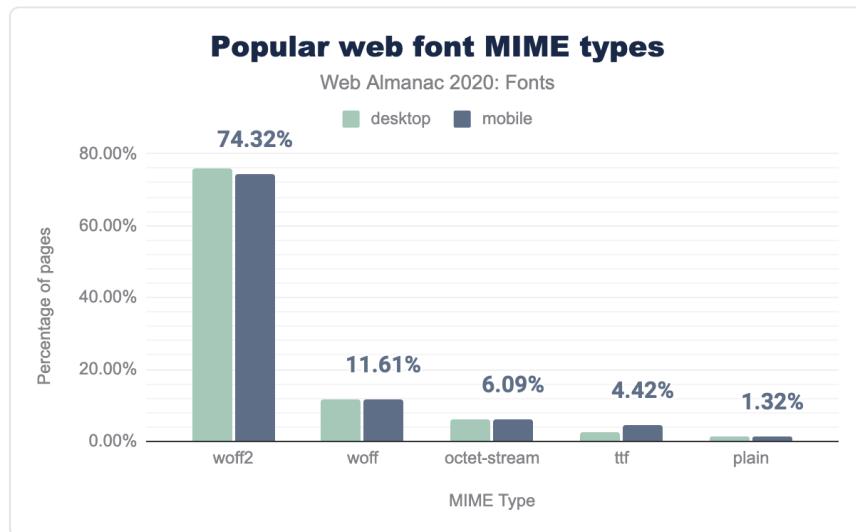


Figure 4.13. Popular web font MIME types.

WOFF is an older, less efficient compression mechanism, but almost universally supported, accounting for an additional 11.6% of fonts served. In almost all cases (Internet Explorer 9-11 being the main exception), serving a font as WOFF is leaving performance on the table, and shows a risk of self-hosting; even if the format choices were optimal at the time of integration, it requires extra effort to update them as browsers improve. Using a hosted service guarantees that the best format is chosen, along with all relevant optimizations.

Ancient versions of Internet Explorer (6-8), which still make about 1.5% of global browser share, only support the EOT format. These don't show up in the top 5 MIME formats, but are necessary for maximum compatibility.

Uncompressed fonts, like OTF and TTF files, are 2-3x larger than compressed, but still make up almost 5% of all fonts served, disproportionately on mobile. If you're serving these, it should be a red flag that optimization is possible.

Popular fonts

Icon fonts are half of the top 10 most popular web fonts, the rest being clean, robust sans-serif typeface designs (Roboto Slab is at #19 and Playfair Display at #26 in this ranking, for debuts of other styles, though serif designs are well represented in the tail of the distribution).

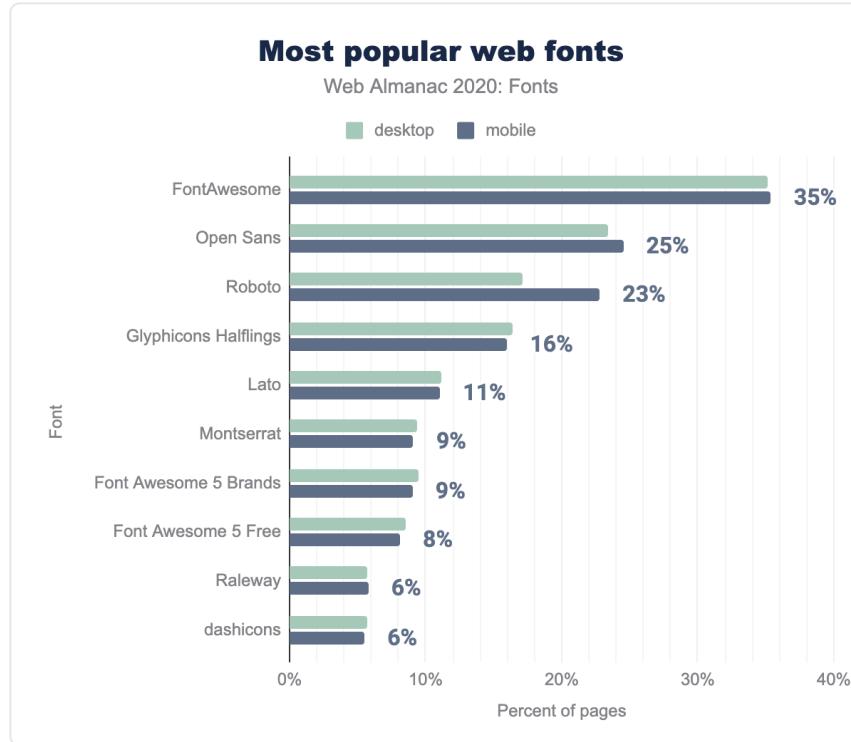


Figure 4.14. Popular typefaces.

A note of caution, in determining the most popular fonts you can get different results depending on measurement methodology. The chart above is based on counting the number of pages that include an `@font-face` rule referencing the named font. That counts multiple styles only once, which arguably weights in favor of single-style fonts.

Color fonts

Color fonts, in one form or other, are supported by most modern browsers, but usage is still close to nonexistent (a total of 755 pages total, the majority of which are in SVG format, which is not supported in Chrome). No doubt part of the problem is the diversity of formats, in fact 4 in widespread use. These come in bitmap and vector flavors. The two bitmap formats are technologically very similar, but SBIX (originally a proprietary Apple format) is not supported in Firefox, while CBDT/CBLC is not supported in Safari.

The COLR vector format is supported on all major modern browsers, but only fairly recently.

The fourth format is embedding SVG in OpenType (not to be confused with SVG fonts), but not supported in Chrome. One drawback of SVG in OpenType is lack of support for font variations, an increasingly important aspect of modern Web design. For this reason, the COLR format is likely to prevail, particularly as support for gradients and clipping is being developed for a future version of COLR. Vector formats are usually much smaller than images, and also scale cleanly to larger sizes, so when COLR arrives with a richer shading model, it could well become popular.

One reason for the poor support of color fonts on the web is that the colors have to be baked into the font files themselves. If you use the same typeface with three different color combinations, near-identical files have to be downloaded three times, and changing a color means reaching for a font editor.

While there is a feature in CSS to override or replace the color palettes in fonts, this has not yet been implemented in browsers, which certainly holds back the ease of deploying color web fonts.

Probably most usage of color fonts is for emoji, but the capability is general purpose and color fonts offer many design possibilities. While color web fonts haven't taken off yet, the underlying technology is heavily used to deliver system emoji, where file format compatibility is much less of an issue.

Browser support is so fragmented that color fonts are not yet tracked by caniuse.com, though there is an issue open for it.

Lots more information about color fonts, including examples, are available at colorfonts.wtf.

Variable fonts

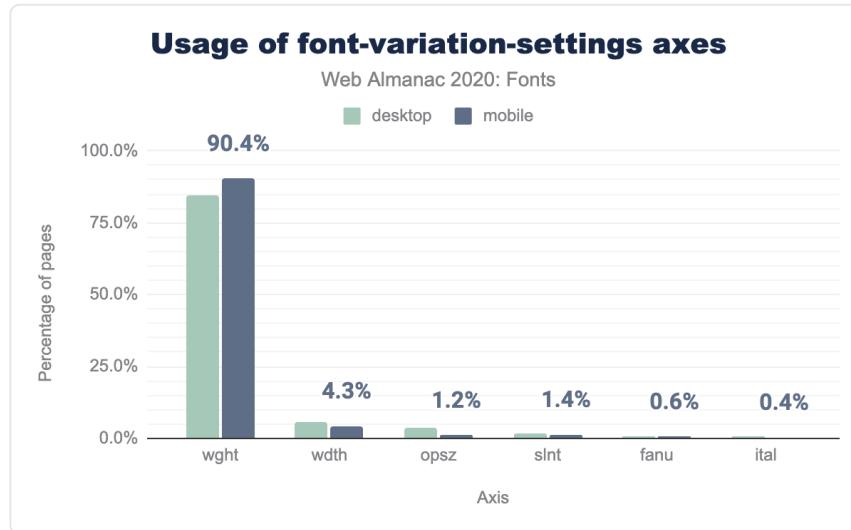


Figure 4.15. Usage of font-variation-settings axes.

Variable fonts are certainly one of the biggest stories this year. They're seen in 8.37% of desktop pages, and 13.84% of mobile. That's up from an average of 1.8% last year, a huge growth factor. It's not hard to see why their popularity is increasing – they offer more design flexibility, and also potentially smaller binary font sizes, especially if multiple styles of the same font are used on the same page.

By far the most commonly used axis is `wght` (which controls weight), at 84.7% desktop and 90.4% mobile. However, `wdth` (width) accounts for approximately 5% of variable font usage. In 2020, Google Fonts began serving 2-axis fonts with both width and weight axes.

It's worth noting that the preferred method is to use `font-weight` and `font-stretch` rather than the lower-level `font-variation-settings` syntax for these two axes as they are completely supported by all browsers that support variable fonts. By setting weight via `font-weight: [number]` and width via `font-stretch: [number]%`, authors provide more appropriate style hints to the browser, which in turn enables better rendering for the end user should the variable font fail to load. This also avoids altering the normal inheritance of styles via the cascade.

The optical size (`opsz`) feature is used for approximately 2% of the variable font usage. This is one to watch, as tuning the appearance of a font to match its intended size of presentation improves the visual refinement in perhaps subtle but very real ways. Usage is also likely to

increase once some current cross-browser and cross-platform uncertainties on how the optical sizes are defined are cleared up. One appealing aspect of the optical size feature is that with the `auto` setting, the variation happens automatically, so the developer gets the benefit of that refinement just by using a font with the `opsz` feature.

There are many potential benefits to using variable fonts. While each included axis increases file size, the tipping point seems to be generally if more than two or three weights of a given typeface are in use, a variable version will likely be similar in total file size or smaller. This is supported by the dramatic increase in variable fonts being served by Google Fonts. Adopting and leveraging variable fonts for more varied design (by using more of the available range of weights and widths) is another. Using a width axis could improve line wrapping on smaller screens, especially with larger headings and longer languages. And with the rise in adoption of alternate light modes, making small adjustments to font-weight when switching modes can improve legibility (see [variablefonts.io](#) for more on usage and implementation).

Towards the future

The performance landscape is changing somewhat, as the advent of cache partitioning reduces the performance benefit from sharing the cache of CDN font resources across multiple sites. The trend of hosting more font assets on the same domain as the site, rather than using a CDN, will probably continue. Even so, services such as Google Fonts are highly optimized, and best practices such as use of `swap` and `preconnect` mitigate much of the impact of the additional HTTP connection.

The use of variable fonts is accelerating greatly, and that trend will no doubt continue, especially as browser and design tool support improve. It's also possible that 2021 will be the year of the color web font; even though the technology has been in place, that certainly hasn't happened yet.

Finally, it's worth mentioning a new concept in web font technology currently being researched by the W3C's Web Font Working Group: Progressive Font Enrichment. PFE is designed as an answer to many of the challenges pointed out in this chapter: addressing performance and user experience when using large glyph count font files (like Arabic or CJK fonts), larger multi-axis or color fonts, or just slow network connectivity environments.

The concept in its simplest terms is that only a portion of a given font file would need to be downloaded in order to render the content on a given page. Subsequent page loads would then deliver a 'patch' to the font file that includes only the glyphs necessary to render each new page. Thus at no time would the user need to download the whole font file at once.

There are various details to work out, including ones that will help ensure privacy and

backwards compatibility—but initial research has been extremely promising and it's hoped this technology will reach the wider web sometime in the next couple years. You can learn more about it in this introduction by Jason Pamental, and read the full Working Group Evaluation Report on the W3C site.

Authors



Raph Levien

 @raphlinus  raphlinus  <https://levien.com>

Raph Levien has been working with fonts for over 35 years, including a PhD from UC Berkeley in font design tools. He is rejoining Google Fonts²⁷ as a font technology researcher, after having co-founded the team in 2010.



Jason Pamental

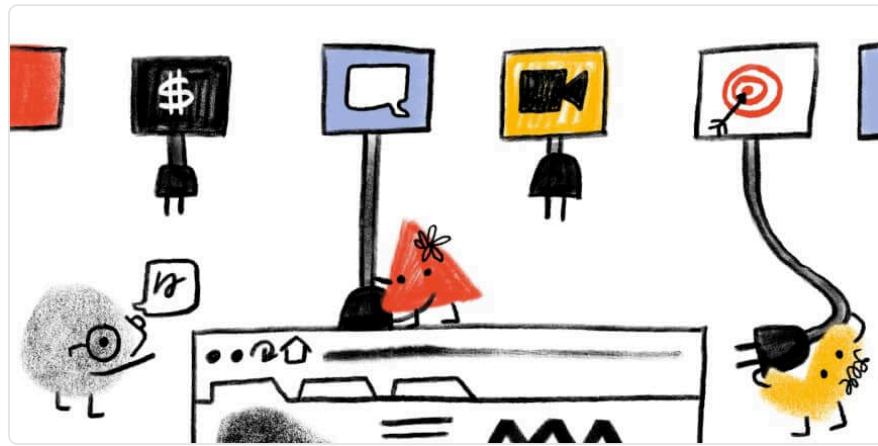
 @jpamental  jpamental  <https://rwt.io>

Designer, tinkerer, typographer. Author of *Responsive Typography*, Invited Expert to the W3C, and 10yrs+ experience focused on better typography on the web.

27. <https://fonts.google.com/>

Part I Chapter 6

Third Parties [UNEDITED]



Written by *Simon Hearne*

Reviewed by *Julia Yang and Shane Exterkamp*

Analyzed by *Max Ostapenko and Paul Calvano*

Introduction

Third-party content is a critical component of most websites today. It powers everything: analytics, live chat, advertising, video sharing and more. Third-party content provides value by taking the heavy lifting off of site owners and allows them to focus on their core competencies.

Many think of third-party content as being JavaScript-based, but the data shows that this is only true for 22% of requests. Third-party content comes in all forms, from images (37%) to audio (0.1%).

In this chapter we will review the prevalence of third-party content and how this has changed since 2019. We will also review: the impact of third-party content on page weight (a good proxy for overall performance impact), scripts that load early in the page lifecycle, the impact of third-party content on browser CPU time, and how open third-parties are with their performance data.

Definitions

"Third Party"

A third party resource is an entity outside the primary site-user relationship. It involves the aspects of the site not directly within the control of the site owner but present, with their approval. For example, the Google Analytics script is a common third-party resource.

Third-party resources are:

- Hosted on a *shared* and *public* origin
- Widely used by a variety of sites
- Uninfluenced by an individual site owner

To match these goals as closely as possible, the formal definition used throughout this chapter for third-party resources is: a resource that originates from a domain whose resources can be found on at least 50 unique pages in the HTTP Archive dataset.

Note that using these definitions, third-party content served from a first-party domain is counted as a first-party content. For example: self-hosting Google Fonts or bootstrap.css is counted as *first-party content*. Similarly, first-party content served from a third-party domain is counted as third-party content. An associated example: First-party images served over a CDN on a third-party domain are considered *third-party content*.

Provider categories

This chapter divides third-party providers into different categories. A brief description is included with each of the categories. The mapping of domain to category can be found in the third-party-web repository.

- Ad - display and measurement of advertisements
- Analytics - tracking site visitor behavior
- CDN - providers that host public shared utilities or private content of their users
- Content - providers that facilitate publishers and host syndicated content
- Customer Success - support and customer relationship management functionality
- Hosting - providers that host the arbitrary content of their users
- Marketing - sales, lead generation, and email marketing functionality
- Social - social networks and their affiliated integrations
- Tag Manager - provider whose sole role is to manage the inclusion of other third parties
- Utility - code that aids the development objectives of the site owner

- Video - providers that host the arbitrary video content of their users
- Other - uncategorized or non-conforming activity

Note on CDNs: The CDN category here includes providers that provide resources on public CDN domains (e.g. bootstrapcdn.com, cdnjs.cloudflare.com, etc.) and does not include resources that are simply served over a CDN. i.e. putting Cloudflare in front of a page would not influence its first-party designation according to our criteria.

Caveats

- All data presented here is based on a non-interactive, cold load. These values could start to look quite different after user interaction.
- The pages are tested from servers in the US with no cookies set, so third-parties requested after opt-in are not included. This will especially affect pages hosted and predominantly served to countries in scope for the General Data Protection Regulation.
- Roughly 84% of all third-party domains by request volume have been identified and categorized. The remaining 16% fall into the "Other" category.

Prevalence

A good starting point for this analysis is to confirm the statement that third-party content is a critical component of most websites today. How many websites use third-party tags, and how many tags do they use?

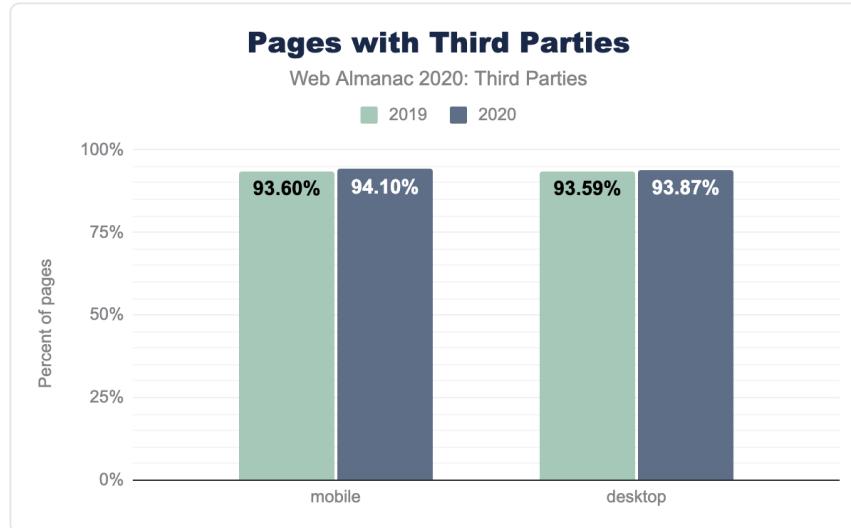


Figure 6.1. Third party content prevalence has grown slightly since 2019

These prevalence numbers show a slight increase on the 2019 results: 93.87% of pages in the desktop crawl had at least one third-party request, the number was slightly higher at 94.10% of pages in the mobile crawl. A brief look into the small number of pages with no third-party content revealed that many were adult sites, some were government domains and some were basic landing / holding pages with little content. It is fair to say that the vast majority of pages have at least one third-party.

The chart below shows the distribution of pages by third-party count. The 10th percentile page has two third-party requests while the median page has 24. Over 10% of pages have more than 100 third-party requests.

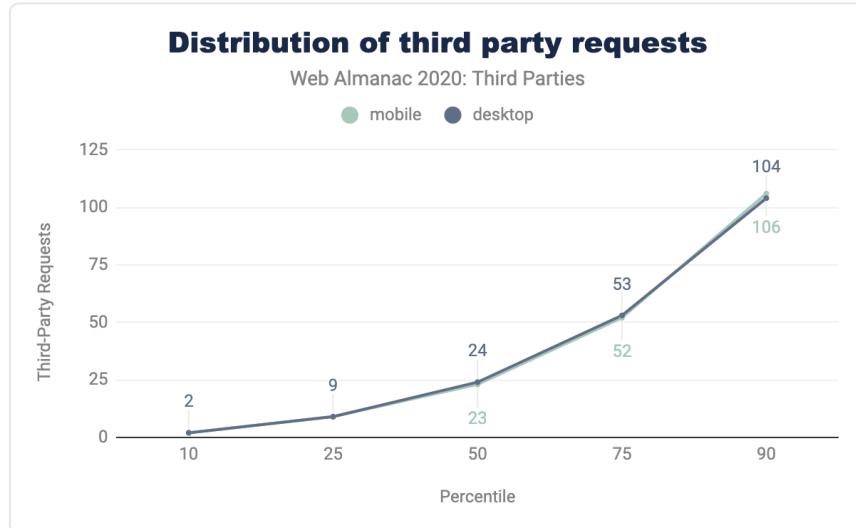


Figure 6.2. The median website has 24 third-party requests

Content-Types

We can break down third-party requests by their content type. This is the reported content-type of the resources delivered from third-party domains.

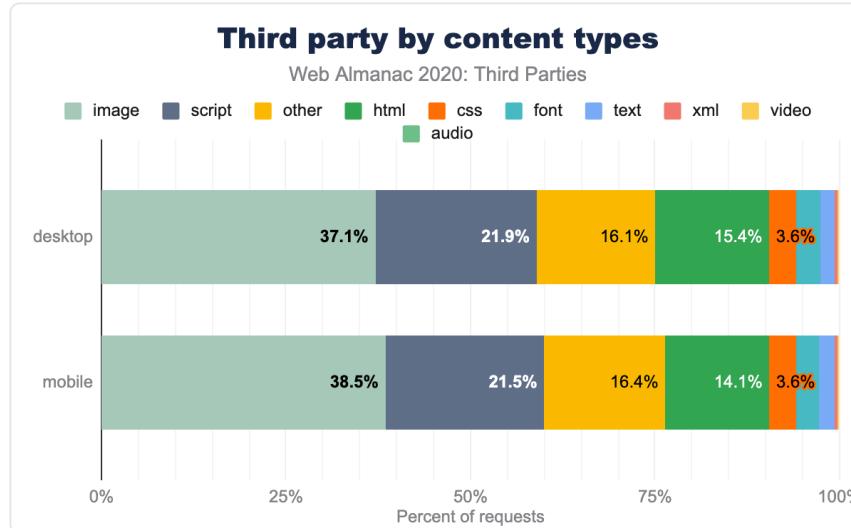


Figure 6.3. Images and JavaScript account for the majority (60%) of third party content

The results show that the major contributors of third-party content are images (38%) and JavaScript (22%), with the next largest contributor being unknown (16%). Unknown is a subset of non-categorized groups such as text/plain as well as responses without a content-type header.

This shows a shift when compared to 2019: relative image content has increased from 33% to 38%, whilst JavaScript content has decreased significantly from 32% to 22%. This reduction is likely due to increased adherence to cookie and data protection regulations, reducing third-party execution until after explicit user opt-in which is out of scope for HTTP Archive test runs.

Third-Party Domains

When we dig further into domains serving third-party content we see that Google Fonts is by far the most common. It is present on more than 7.5% of mobile pages tested. While fonts only account for around 3% of third-party content, almost all of these are delivered by the Google Fonts service. If your page uses Google Fonts, make sure to follow best practices to ensure the best possible user experience.

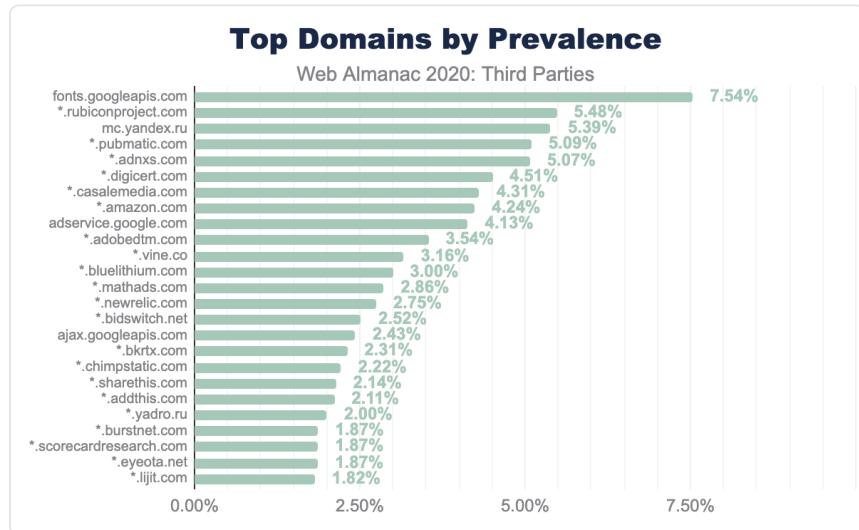


Figure 6.4. The most prevalent domains are font foundries, advertising, social media and JavaScript CDNs

The next four most common domains are all advertising providers, they may not be requested directly by your page but through a complex chain of redirects initiated by another advertising network.

The sixth most common domain is digicert.com. Calls to digicert.com are generally OCSP revocation checks due to TLS certificates not having OCSP stapling enabled, or the use of Extended Validation (EV) certificates which prevent pinning of intermediate certificates. This number is exaggerated in HTTP Archive due to all page loads being effectively first-time visitors - OCSP responses are generally valid for seven days in real-world browsing. See this blog post to read more on this issue.

Further down the list at 2.43% is ajax.googleapis.com, Google's Hosted Libraries project. Whilst loading a library such as jQuery from a hosted service is easy, the additional cost of a connection to a third-party domain may have a negative impact on performance. It is best to host all critical JavaScript and CSS on the root domain, if possible. There is also now no cache benefit to using a shared CDN resource, as all major browsers partition caches by page. Harry Roberts has written a detailed blog post on how to host your own static assets.

Page Weight Impact

Heaviest third-parties

We can extract the largest third-parties by the median page weight impact, i.e. how many bytes they bring to the pages they are on. The results are interesting as this does not take into account how popular the third-parties are, just their impact in bytes.

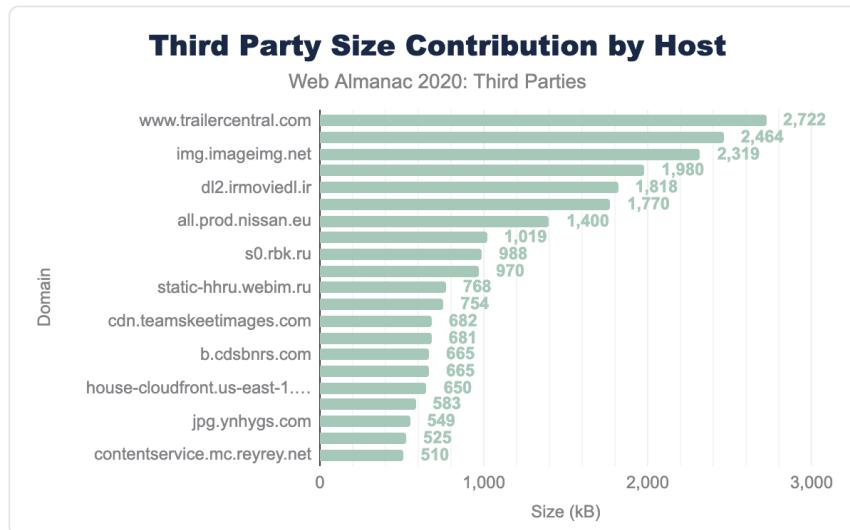


Figure 6.5. Media providers result in the largest contribution to page size

The top contributors of page weight are generally media content providers, such as image and video hosting. Vidazoo, for example, results in a median page weight impact of about 2.5MB. The `inventory.vidazoo.com` domain provides video hosting, so a median page with this third-party has an extra 2.5MB of media content!

A simple method to reduce this impact is to defer video loading until a user interacts with the page, so that the impact is reduced for those visitors that never consume the video.

We can take this analysis further to produce a distribution of total page size (in bytes downloaded for all resources) by third-party category presence. This chart shows that the presence of most third-party categories does not have a noticeable impact on total page size: this would be visible as a divergence in the plots. A notable exception to this is Advertising (in black) which shows a very small relationship with page size, indicating that advertisement requests do not add significant weight to pages. This is likely because many of these requests are small redirects, the median is only 420 bytes. We see similar low impact for tag managers,

and analytics.

On the other end of the spectrum, the categories CDN, Content and Hosting all represent strong relationship with total page weight. This indicates that sites using hosted services are generally larger in page weight.

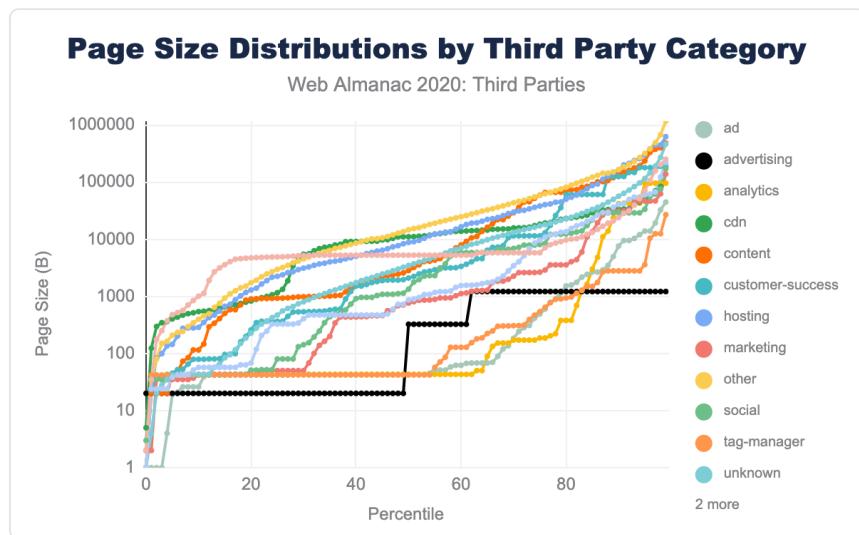


Figure 6.6. Advertising third parties have little impact on page size, CDN and Hosting has a significant impact

Cacheability

Some third-party responses should always be cached. Media such as images and videos served by a third-party, or JavaScript libraries are good candidates. The results show that overall two-thirds of third-party requests are served with a valid caching header such as `cache-control`.

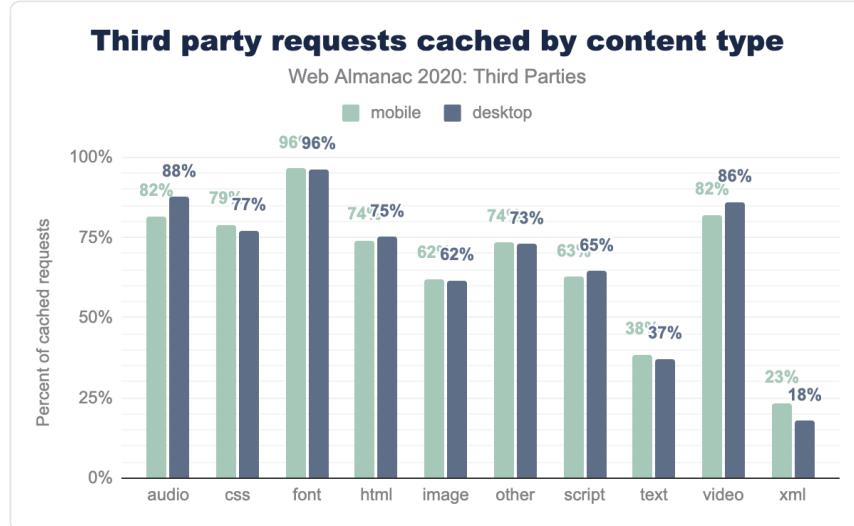


Figure 6.7. Font assets are the most likely to be cached, with text and xml resources least likely.

Breaking down by response type highlights some common offenders: xml and text responses are less likely to be cacheable. Surprisingly, less than two-thirds of images served by third-parties are cacheable. On further inspection, this is due to the use of tracking 'pixels' which are returned as non-cacheable zero-size image responses.

Large redirects

Many third-parties result in redirect responses, i.e. HTTP status codes 3XX. These occur due to the use of vanity domains or to share information across domains through request headers. This is especially true for advertising networks. Large redirect responses are an indication of a misconfiguration, as the response should be around 340B for a valid `Location` response header plus overheads. The chart below shows the distribution of body size for all third-party redirects in the HTTP Archive.

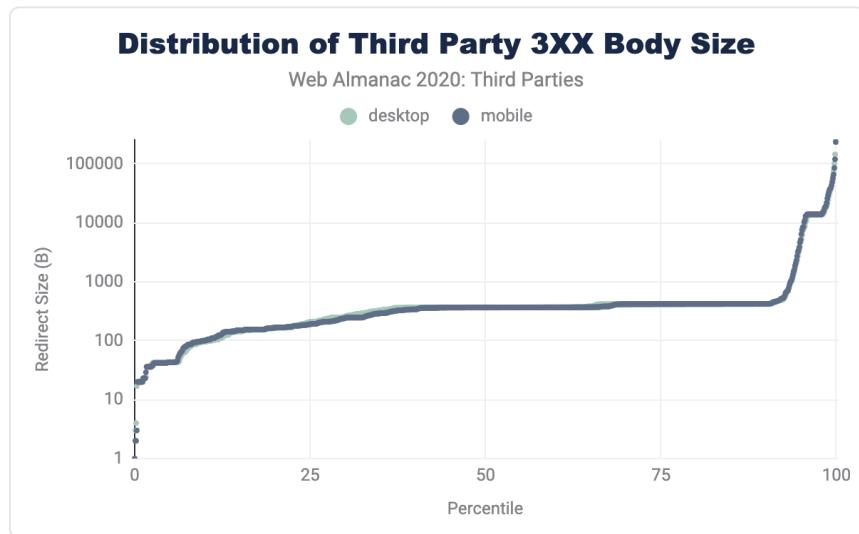


Figure 6.8. Most redirects are under 420B, the 99th percentile is over 30kB!

The results show that the majority of 3XX responses are small: the 90th percentile is 420B, i.e. 90% of 3XX responses are 420 bytes or smaller. The 95th percentile is 6.5kB, the 99th is 36kB and the 99.9th is over 100kB! Whilst redirects may seem innocuous, 100kB is an unreasonable amount of bytes over the wire for a response that simply leads to another response.

Early-loaders

Scripts that load late in the page will have an impact on total page load duration and page weight, but might have no impact on the user experience. Scripts that load early in the page, however, will potentially cannibalize bandwidth for critical first-party resources and are more likely to interfere with the page load. This can have a detrimental impact on performance metrics and user experience.

The chart below shows the percentage of requests that load early, by device type and third-party category. The three stand-out categories are CDN, Hosting and Tag Managers: all of which tend to deliver JavaScript that is requested in the head of a document. Advertising resources are least likely to load early in the page, this is due to advertisement network requests generally being asynchronous scripts run after page load.

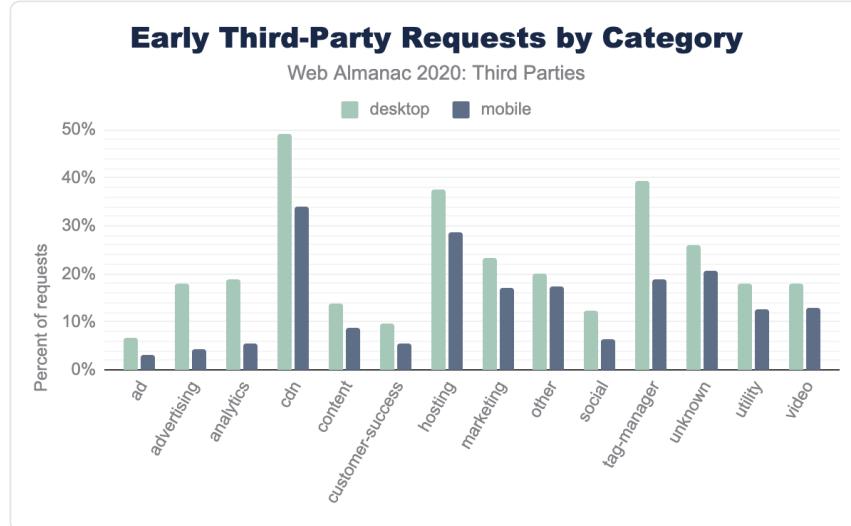


Figure 6.9. Public CDN resources are most likely to be downloaded before DOM Content Loaded, with Social, Advertising and Customer Success most likely to be loaded later.

CPU Impact

Not all bytes on the web are equal: a 500kB image may be far easier for a browser to process than a 500kB compressed JavaScript bundle, which inflates to 1.8MB of client-side code! The impact of third-party scripts on CPU time can be far more critical than the additional bytes or time spent on the network.

We can correlate the presence of third-party categories with the total CPU time on the page, this allows us to estimate the impact of each third-party category on CPU time.

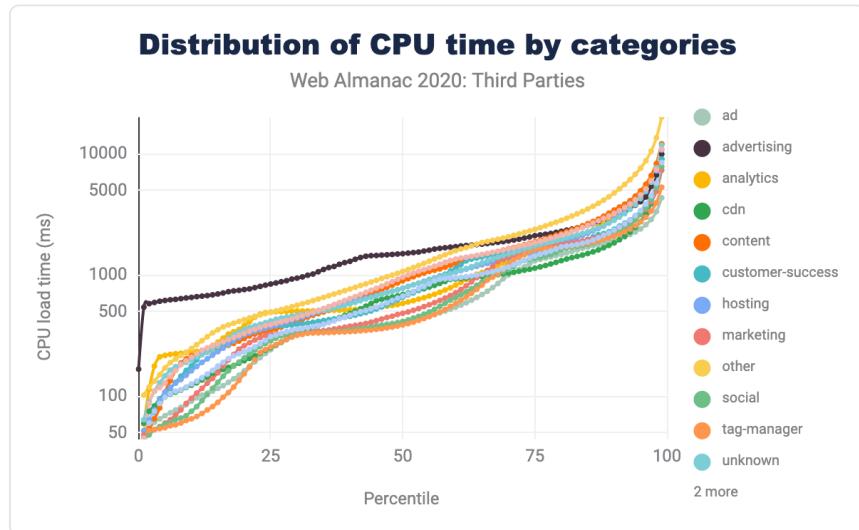


Figure 6.10. Pages with advertising third parties are more likely to have a high CPU load time.

This chart shows the probability density function of total page CPU time by the third-party categories present on each page. The median page is at 50 on the percentile axis. The data shows that all third-party categories follow a similar pattern, with the median page between 400 - 1,000ms CPU time. The outlier here is advertising (in black): if a page has advertising tags it is much more likely to have high CPU usage during page load. The median page with advertising tags has a CPU load time of 1,500ms, compared to 500ms for pages without advertising. The high CPU load time at the lower percentiles indicates that even the fastest sites are impacted significantly by the presence of third-parties categorized as advertising.

Other

Timing-Allow-Origin prevalence

The Resource Timing API allows website owners to measure the performance of individual resources via JavaScript. This data is, by default, extremely limited for cross-origin resources like third-party content. There are legitimate reasons for not providing this timing information such as responses that vary by authentication state: e.g. a website owner may be able to determine if a visitor is logged into a Facebook by measuring the response size of a widget request. For most third-party content, though, setting the `timing-allow-origin` header is an act of transparency to allow the hosting website to track performance and size of their third-party content.

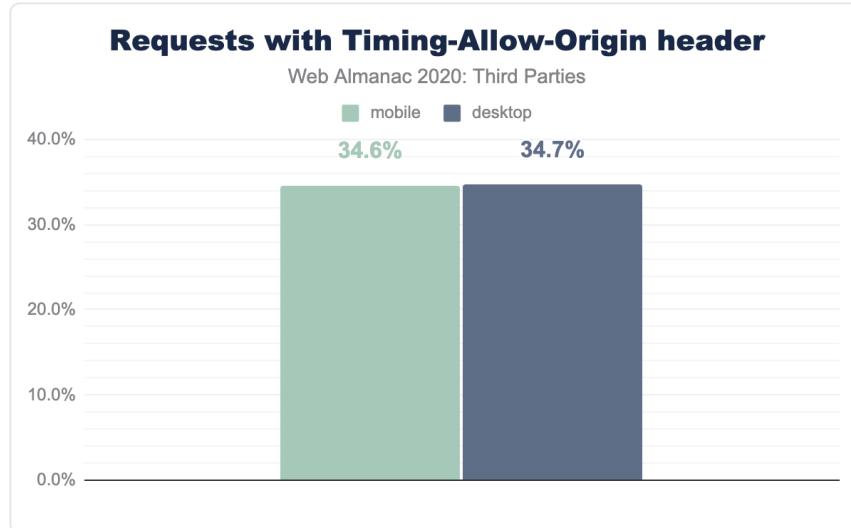


Figure 6.11. Less than 35% of third party responses are served with a timing-allow-origin header

The results in HTTP Archive show that only one third of third-party responses expose detailed size and timing information to the hosting website.

Repercussions

We know that adding arbitrary JavaScript to our sites introduces risks to both site speed and security. Site owners must be diligent to balance the value of the third-party scripts they include with the speed penalty they may bring, and use modern features such as subresource integrity and content security policy to maintain a strong security posture. See the Security chapter for more detail on these and other browser security features.

Conclusion

One of the surprises in the data from 2020 is the drop in relative JavaScript requests: from 32% of the total to just 22%. It is unlikely that the actual amount of JavaScript on the web has decreased this significantly; it is more likely that websites are implementing consent management - so that most dynamic third-party content is only loaded on user opt-in. This opt-in process could be managed by a consent management platform (CMP) in some cases. The third-party database does not yet have a category for CMPs, but this would be a good analysis for the 2021 Web Almanac.

Advertising requests appear to have an increased impact on CPU time, the median page with advertising scripts consume three times as much CPU as those without. Interestingly though, advertising scripts are not correlated with increased page weight. This makes it even more important to evaluate the total impact of third-party scripts on the browser, not just request count and size.

While third-party content is critical to many websites, auditing the impact of each provider is critical to ensure that they do not significantly impact user experience, page weight or CPU utilization. There are often self-hosting options for the top contributors to third-party weight, this is especially worth considering as there is now no caching benefit to using shared assets:

- Google Fonts allows self-hosting the assets
- JavaScript CDNs can be replaced with self-hosted assets
- Experimentation scripts can be self-hosted, e.g. Optimizely

Author

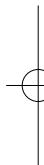
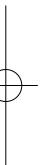


Simon Hearne

 @simonhearne  simonhearne  <https://simonhearne.com>

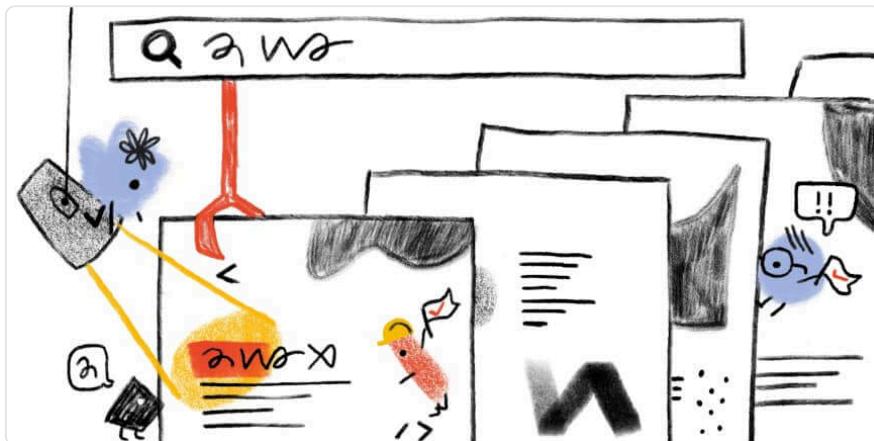
Simon is a web performance architect, he is passionate about helping deliver a faster and more accessible web. You can find him tweeting @SimonHearne and blogging at simonhearne.com²⁸.

28. <https://simonhearne.com>



Part II Chapter 7

SEO



Written by Aleyda Solis, Michael King, and Jamie Indigo

Reviewed by Nate Dame, Catalin Rosu, Dave Sottimano, Dave Smart, Dustin Montgomery, Sawood Alam, and Barry Pollard

Analyzed by Tony McCreath and Antoine Eripert

Introduction

Search Engine Optimization (SEO) is the practice of optimizing websites' technical configuration, content relevance, and link popularity to make their information easily findable and more relevant to fulfill users' search needs. As a consequence, websites improve their visibility in search engines' results for relevant user queries regarding their content and business, growing their traffic, conversions, and profits.

Despite its complex multidisciplinary nature, in recent years SEO has evolved to become one of the most popular digital marketing strategies and channels.

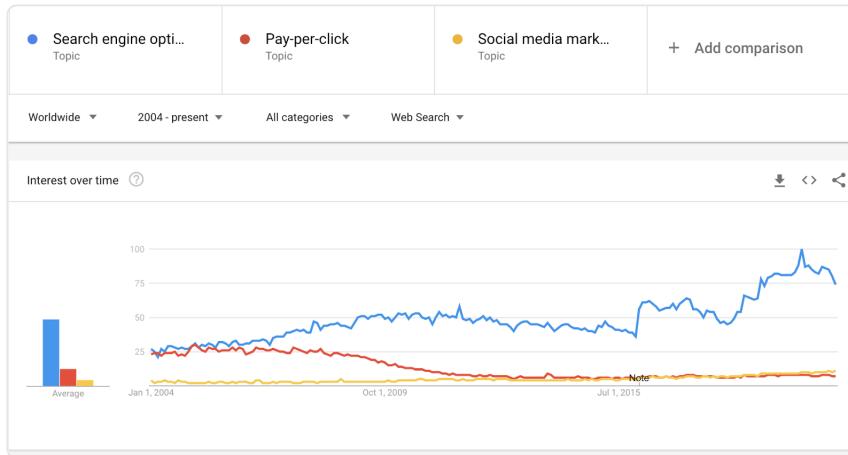


Figure 7.1. Google Trends comparison of SEO versus pay-per-click and social media marketing.

The goal of the Web Almanac's SEO chapter is to identify and assess main elements and configurations that play a role in a website's organic search optimization. By identifying these elements, we hope that websites can leverage our findings to improve their ability to be crawled, indexed, and ranked by search engines. In this chapter, we provide a snapshot of their status in 2020 and a summary of what has changed since 2019.

It is important to note that this chapter is based on analysis from Lighthouse, the Chrome UX Report, as well as raw and rendered HTML elements from the HTTP Archive crawl. In the case of the HTTP Archive and Lighthouse, it is limited to the data identified from websites' home pages only, not site-wide crawls. We have taken this into consideration when doing assessments. Keeping this distinction in mind is important when drawing conclusions from our results. You can learn more about it on our Methodology page.

Let's go through this year's organic search optimization main findings.

Fundamentals

This section features the optimization-related findings of the web configurations and elements that make up the foundation for search engines to correctly crawl, index, and rank websites to provide users the best results for their queries.

Crawlability and indexability

Search engines use web crawlers (also called spiders) to discover new or updated content from

websites, browsing the web by following links between pages. Crawling is the process of looking for new or updated web content (whether web pages, images, videos, etc.).

Search crawlers discover content by following links between URLs, as well as using additional sources that website owners can provide, like the generation of XML sitemaps, which are lists of URLs that a website's owner wants search engines to index, or through direct crawl requests via search engines tools, like Google's Search Console.

Once search engines access web content they need to *render*—similar to what web browsers do—and index it. Search engines will then analyze and catalog the identified information, trying to understand it as users do, to ultimately store it in its *index*, or web database.

When users enter a query, search engines search their index to find the best content to display on the search results pages to answer their queries, using a variety of factors to determine which pages are shown before others.

For websites looking to optimize their visibility in search results, it is important to follow certain crawlability and indexability best practices: correctly configuring `robots.txt`, robots `meta` tags, `X-Robots-Tag` HTTP headers, and canonical tags, among others. These best practices help search engines in accessing web content more easily and indexing them more accurately. A thorough analysis of these configurations is provided in the following sections.

`robots.txt`

Located at the root of a site, a `robots.txt` file is an effective tool in controlling which pages a search engine crawler should interact with, how quickly to crawl them, and what to do with the discovered content.

Google formally proposed making `robots.txt` an official internet standard in 2019. The June 2020 draft includes clear documentation on technical requirements for the `robots.txt` file. This has prompted more detailed information about how search engine crawlers should respond to non-standard content.

A `robots.txt` file must be plain text, encoded in UTF-8, and respond to requests with a 200 HTTP status code. A malformed `robots.txt`, a 4XX (client error) response, or more than five redirects are interpreted by search engine crawlers as a *full allow*, meaning all content may be crawled. A 5XX (server error) response is understood as a *full disallow*, meaning no content may be crawled. If the `robots.txt` is unreachable for more than 30 days, Google will use the last cached copy of it, as described in their specifications.

Overall, 80.46% of mobile pages responded to `robots.txt` with a 2XX response. Of these, 25.09% were not recognized as valid. This has slightly improved over 2019, when it was found

that 27.84% of mobile sites had a valid `robots.txt`.

Lighthouse, the data source for testing `robots.txt` validity, introduced a `robots.txt` audit as part of the v6 update. This inclusion highlights that a successfully resolved request does not mean that the cornerstone file will be able to provide the necessary directives to web crawlers.

Response Code	Mobile	Desktop
2XX	80.46%	79.59%
3XX	0.01%	0.01%
4XX	17.67%	18.64%
5XX	0.15%	0.12%
6XX	0.00%	0.00%
7XX	0.15%	0.12%

Figure 7.2. `robots.txt` response codes.

In addition to similar status code behavior, `Disallow` statement use was consistent between mobile and desktop versions of `robots.txt` files.

The most prevalent `User-agent` declaration statement was the wildcard, `User-agent: *`, appearing on 74.40% of mobile and 73.16% of desktop `robots.txt` requests. The second most prevalent declaration was `adsbot-google`, appearing in 5.63% of mobile and 5.68% of desktop `robots.txt` requests. Google AdsBot disregards wildcard statements and must be specifically named as the bot checks web page and app ad quality across devices.

The most frequently used directives focused on search engines and their paid marketing counterparts. SEO tools Ahref and Majestic were in the top five `Disallow` statements for both devices.

% of robots.txt		
User-agent	Mobile	Desktop
*	74.40%	73.16%
adsbot-google	5.63%	5.68%
mediapartners-google	5.55%	3.83%
mj12bot	5.49%	5.30%
ahrefsbot	4.80%	4.66%

Figure 7.3. robots.txt User-agent directives.

When analyzing the usage of the `Disallow` statement in `robots.txt` by using Lighthouse-powered data of over 6 million sites, it was found that 97.84% of them were completely crawlable, with only 1.05% using a `Disallow` statement.

An analysis of the `robots.txt` `Disallow` statement usage along the meta robots `indexability` directives was also done, finding 1.02% of the sites including a `Disallow` statement along indexable pages featuring a meta robots `index` directive, with only 0.03% of sites using the `Disallow` statement in `robots.txt` along `noindexed` pages via the meta robots `noindex` directive.

The higher usage of the `Disallow` statement on indexable pages than noindexed ones is notable as Google documentation states that site owners should not use `robots.txt` as a means to hide web pages from Google Search, as internal linking with descriptive text could result in the page being indexed without a crawler visiting the page. Instead, site owners should use other methods, like a `noindex` directive via meta robots.

Meta robots

The `robots` meta tag and `X-Robots-Tag` HTTP header are an extension of the proposed Robots Exclusion Protocol (REP), which allows directives to be configured at a more granular level. Directive support varies by search engine as REP is not yet an official internet standard.

Meta tags were the dominant method of granular execution with 27.70% of desktop and 27.96% of mobile pages using the tag. `X-Robots-Tag` directives were found on 0.27% and 0.40% of desktop and mobile, respectively.

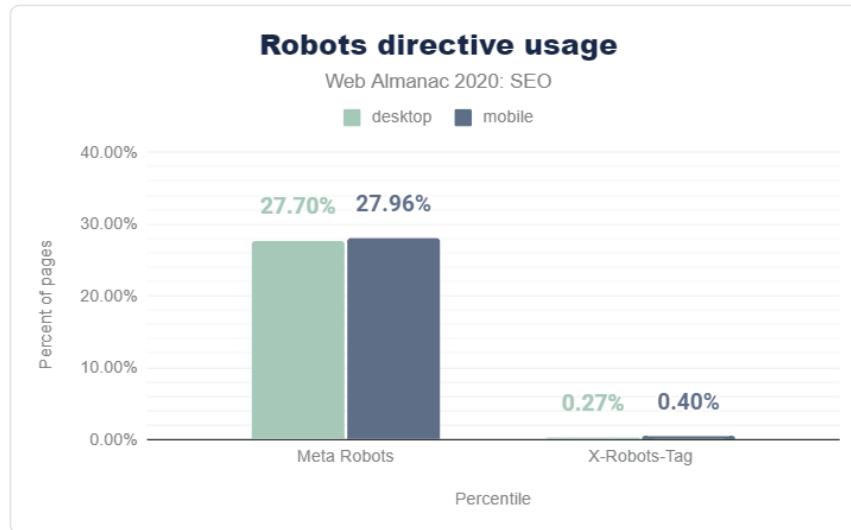


Figure 7.4. Usage of meta robots and `X-Robots-Tag` directives.

When analyzing the usage of the meta robots tag in Lighthouse tests, 0.47% of crawlable pages were found to be `noindexed`. 0.44% of these pages used a `noindex` directive and did not disallow crawling of the page in the `robots.txt`.

The combination of `Disallow` within `robots.txt` and `noindex` directive in meta robots were found on only 0.03% of pages. While this method offers *belt and suspenders* redundancy, a page must not be blocked by a `robots.txt` file in order for an on-page `noindex` directive to be effective.

Interestingly, rendering changed the meta robots tag in 0.16% of pages. While there is no inherent issue with using JavaScript to add a meta robots tag to a page or change its content, SEOs should be judicious in execution. If a page loads with a `noindex` directive in the meta robots tag before rendering, search engines won't run the JavaScript that changes the tag value or index the page.

Canonicalization

Canonical tags, as described by Google, are used to specify to search engines which is the preferred canonical URL version to index and rank for a page—the one that is considered to be better representative of it—when there are many URLs featuring the same or very similar content. It is important to note that:

- The canonical tag configuration is used along with other signals to select the

canonical URL of a page; it is not the only one.

- Although self-referencing canonical tags are sometimes used, these aren't a requirement.

In last year's chapter, it was identified that 48.34% of mobile pages were using a canonical tag. This year the number of mobile pages featuring a canonical tag has grown to 53.61%.

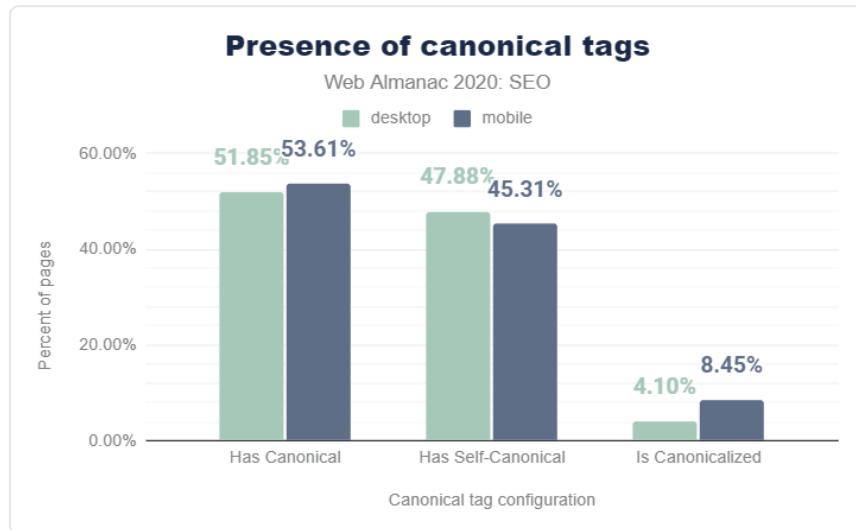


Figure 7.5. Usage of canonical tags.

When analyzing this year's mobile pages canonical tag configuration, it was detected that 45.31% of them were self-referential and 8.45% were pointing to different URLs as the canonical ones.

On the other hand, 51.85% of the desktop pages were found to be featuring a canonical tag this year, with 47.88% being self-referential and 4.10% pointing to a different URL.

Not only do mobile pages include more canonical tags than desktop ones (53.61% versus 51.85%), there are relatively more mobile homes pages canonicalizing to other URLs than their desktop counterparts (8.45% vs. 4.10%). This could be explained by the usage of an independent (or separate) mobile web version by some sites that need to canonicalize to their desktop URLs alternates.

Canonical URLs can be specified through different methods: by using the canonical link via the HTTP headers or the HTML `head` of a page, or by submitting them in XML sitemaps. When analyzing which is the most popular canonical link implementation method, it was found that only 1.03% of desktop pages and 0.88% of mobile ones are relying on the HTTP headers for

their implementation, meaning that canonical tags are prominently implemented via the HTML `head` of a page.

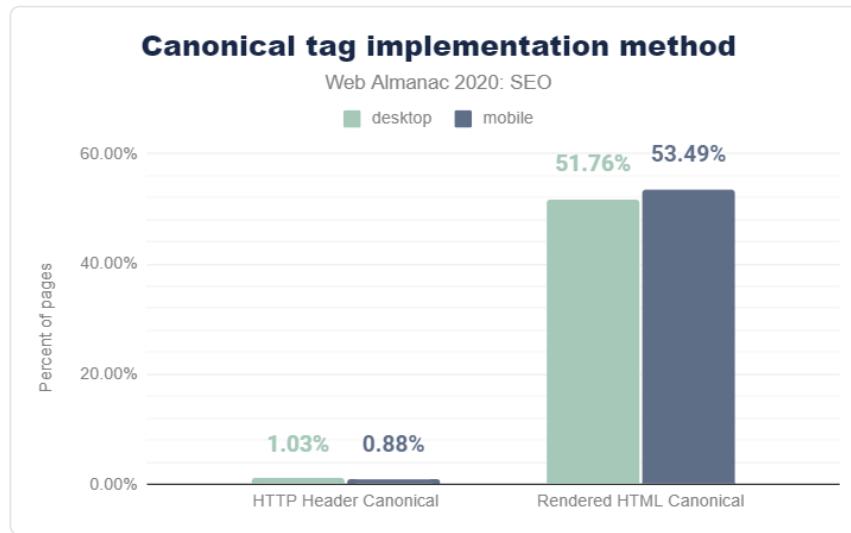


Figure 7.6. Usage of HTTP header and HTML `head` canonicalization methods.

When analyzing the canonical tag implemented in the raw HTML versus those relying on client-side JavaScript rendering, we identified that 0.68% of the mobile pages and 0.54% of the desktop ones include a canonical tag in the rendered but not the raw HTML. This means that there's only a very small number of pages that are relying on JavaScript to implement canonical tags.

On the other hand, in 0.93% of the mobile pages and 0.76% of the desktop ones, we saw canonical tags implemented via both the raw and the rendered HTML with a conflict happening between the URL specified in the raw versus the rendered HTML of the same pages. This can generate indexability issues as mixed information is sent to search engines about which is the canonical URL for the same page.

A similar conflict can be found with the different implementation methods, with 0.15% of the mobile pages and 0.17% of the desktop ones showing conflicts between the canonical tags implemented via their HTTP headers and HTML `head`.

Content

The primary purpose that both search engines and Search Engine Optimization serve is to give visibility to content that users need. Search engines extract features from pages to determine

what the content is about. In that way, the two are symbiotic. The features extracted align with signals that indicate relevance and inform ranking.

To understand what search engines are able to effectively extract, we have broken out the components of that content and examined the incidence rate of those features between the mobile and desktop contexts. We also reviewed the disparity between mobile and desktop content. The mobile and desktop disparity is especially valuable because Google has moved to *mobile-first indexing (MFI)* for all new sites and, as of March of 2021, will move to a *mobile-only index* wherein content that does not appear within the mobile context will not be evaluated for ranking.

Rendered versus non-rendered text content

The usage of Single Page Application (SPA) JavaScript technologies has exploded with the growth of the web. This design pattern introduces difficulties for search engine spiders because both the execution of JavaScript transformations at runtime and user interactions with the page after load can cause additional content to appear or be rendered.

Search engines encounter pages through its crawling activity, but may or may not choose to implement a second step of rendering a page. As a result, there may be disparities between the content that a user sees and the content that a search engine indexes and considers for rankings.

We assessed word count as a heuristic of that disparity.

Values	Desktop	Mobile	Difference
Raw	360	312	-13.33%
Rendered	402	348	-13.43%
Difference	11.67%	11.54%	

Figure 7.7. Comparison of the median number of raw and rendered words per desktop and mobile page.

This year, the median desktop page was found to have 402 words and the mobile page had 348 words. While last year, the median desktop page had 346 words, and the median mobile page had a slightly lower word count at 306 words. This represents 16.2% and 13.7% growth respectively.

We found that the median desktop site features 11.67% more words when rendered than it does on an initial crawl of its raw HTML. We also found that the median mobile site displays

13.33% less text content than its desktop counterpart. The median mobile site also displays 11.54% more words when rendered than its raw HTML counterpart.

Across our sample set, there are disparities across the combination of mobile/desktop and rendered/non-rendered. This suggests that although search engines are continually improving in this area, most sites across the web are missing out on opportunities to improve their organic search visibility through a stronger focus on ensuring their content is available and indexable. This is also a concern because the lion's share of available SEO tools do not crawl in the above combination of contexts and automatically identify this as an issue.

Headings

Heading elements (H1 - H6) act as a mechanism to visually indicate structure in a page's content. Although these HTML elements don't carry the weight they used to in search rankings, they still act as a valuable way to structure pages and signal other elements in the search engine results pages (SERPs) like *featured snippets* or other extraction methods that align with Google's new passage indexing.

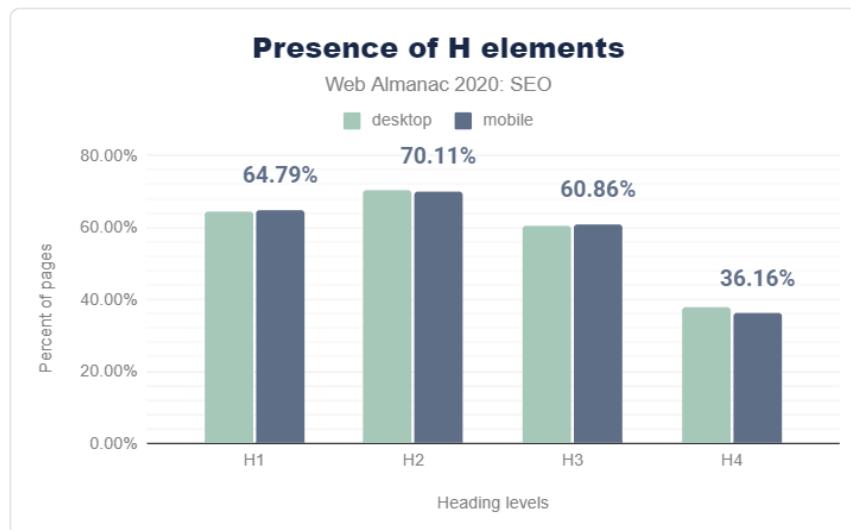


Figure 7.8. Usage of heading levels 1 through 4, including empty headings.

Over 60% of pages feature H1 elements (including empty ones) in both the mobile and desktop contexts.

These numbers hover around 60%+ through H2 and H3. The incidence rate of H4 elements is lower than 4%, suggesting that the level of specificity is not required for most pages or the

developers style other headings elements differently to support the visual structure of the content.

The prevalence of more `H2` elements than `H1`'s suggests that fewer pages are using multiple `H1`'s.

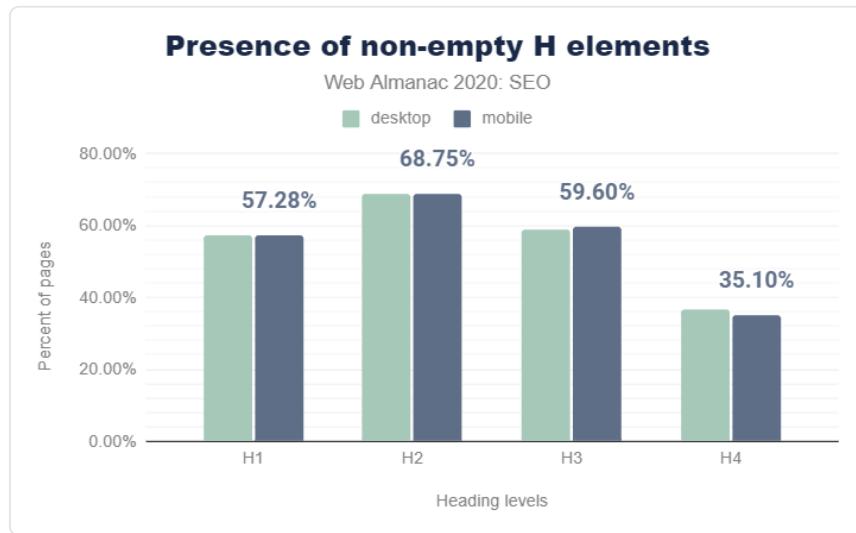


Figure 7.9. Usage of heading levels 1 through 4, excluding empty headings.

In reviewing the adoption of non-empty heading elements, we found that 7.55% of `H1`, 1.4% of `H2`, 1.5% of `H3`, and 1.1% of `H4` elements feature no text. One possible explanation for these low results is that those portions are used for styling the page or are the result of coding mistakes.

You can learn more about the usage of headings in the Markup chapter, including the misuse of non-standard `H7` and `H8` elements.

Structured data

Over the course of the past decade, search engines, particularly Google, have continued to push towards becoming the presentation layer of the web. These advancements are partially driven by their improved ability to extract information from unstructured content (e.g., passage indexing) and the adoption of semantic markup in the form of *structured data*. Search engines have encouraged content creators and developers to implement structured data to give more visibility to their content within components of search results.

In a move from "strings to things", search engines have agreed upon a broad vocabulary of objects in support of marking up a variety of people, places, and things within web content. However, only a subset of that vocabulary triggers inclusion within search results components. Google specifies those that they support and how they're displayed in their search gallery, and provides a tool to validate their support and implementation.

As search engines evolve to reflect more of these elements in search results, the incidence rates of the different vocabularies change across the web.

As part of our examination, we took a look at the incidence rates of different types of structured markup. The available vocabularies include RDFa and schema.org, which come in both the microformats and JSON-LD flavors. Google has recently dropped the support for data-vocabulary, which was primarily used to implement breadcrumbs.

JSON-LD is generally considered to be the more portable and easier to manage implementation and so it has become the preferred format. As a result, we see that JSON-LD appears on 29.78% of mobile pages and 30.60% of desktop pages.

Format	Mobile	Desktop
JSON-LD	29.78%	30.60%
Microdata	19.55%	17.94%
RDFa	1.42%	1.63%
Microformats2	0.10%	0.10%

Figure 7.10. Usage of each structured data format.

We find that the disparity between mobile and desktop continues with this type of data.

Microdata appeared on 19.55% of mobile pages and 17.94% of desktop pages. RDFa appeared on 1.42% of mobile pages and 1.63% of desktop pages.

Rendered versus non-rendered structured data

We found that 38.61% of desktop pages and 39.26% of mobile pages feature JSON-LD or microformat structured data in the raw HTML, while 40.09% of desktop pages and 40.97% of mobile pages feature structured data in the rendered DOM.

When reviewing this in more detail, we found that 1.49% of desktop pages and 1.77% of mobile pages only featured this type of structured data in the rendered DOM due to JavaScript transformations, relying in search engines JavaScript execution capabilities.

Finally, we found that 4.46% of desktop pages and 4.62% of mobile pages feature structured data that appears in the raw HTML and is subsequently changed by JavaScript transformations in the rendered DOM. Depending on the type of changes applied to the structured data configuration, this could generate mixed signals for search engines when rendering them.

Most prevalent structured data objects

As seen last year, the most prevalent structured data objects remain to be `WebSite`, `SearchAction`, `WebPage`, `Organization`, and `ImageObject`, and their usage has continued to grow:

- `WebSite` has grown 9.37% on desktop and 10.5% on mobile
- `SearchAction` has grown 7.64% on both desktop and mobile
- `WebPage` has grown on desktop 6.83% and 7.09% on mobile
- `Organization` has grown on desktop 4.75% and 4.98% on mobile
- `ImageObject` has grown 6.39% on desktop and 6.13% on mobile

It should be noted that `WebSite`, `SearchAction` and `Organization` are all typically associated with home pages, so this highlights the bias of the dataset and does not reflect the bulk of structured data implemented on the web.

In contrast, despite the fact that reviews are not supposed to be associated with home pages, the data indicates that `AggregateRating` is used on 23.9% on mobile and 23.7% on desktop.

It's also interesting to see the growth of the `VideoObject` to annotate videos. Although YouTube videos dominate video search results in Google, the usage of `VideoObject` grew 30.11% on desktop and 27.7% on mobile.

The growth of these objects is a general indication of increased adoption of structured data. There's also an indication of what Google gives visibility within search features increases the incidence rates of lesser used objects. Google announced the `FAQPage`, `HowTo`, and `QAPage` objects as visibility opportunities in 2019 and they sustained significant year-over-year growth:

- `FAQPage` markup grew 3,261% on desktop and 3,000% on mobile.
- `HowTo` markup grew 605% on desktop and 623% on mobile.
- `QAPage` markup grew 166.7% on desktop and 192.1% on mobile.

Again, it's important to note that this data might not be representative of their actual level of growth, since these objects are usually placed on internal pages.

The adoption of structured data is a boon for the web as extracting data is valuable to a wealth of use cases. We expect this to continue to grow as search engines expand their usage and as it begins to power applications beyond web search.

Metadata

Metadata is an opportunity to describe and explain the value of the content on the other side of the click. While page titles are believed to be weighed directly in search rankings, meta descriptions are not. Both elements can encourage or discourage a user to click or not click based on their contents.

We examined these features to see how pages are quantitatively aligning with best practices to drive traffic from organic search.

Titles

The page title is shown as the anchor text in search engine results and is generally considered one of the most valuable on-page elements that impacts a page's ability to rank.

When analyzing the usage of the `title` tag, we found that 99% of desktop and mobile pages have one. This represents a slight improvement since last year, when 97% of mobile pages had a `title` tag.

The median page features a page title that is six words long. There is no difference in the word count between the mobile and desktop contexts within our dataset. This suggests that the page title element is an element that is not modified between different page template types.



Figure 7.11. Distribution of the number of words per page title.

The median page title character count is 38 characters on both mobile and desktop. Interestingly, this is up from 20 characters on desktop and 21 characters on mobile from last year's analysis. The disparity between the contexts has disappeared year-over-year except within the 90th percentile wherein there is a one character difference.

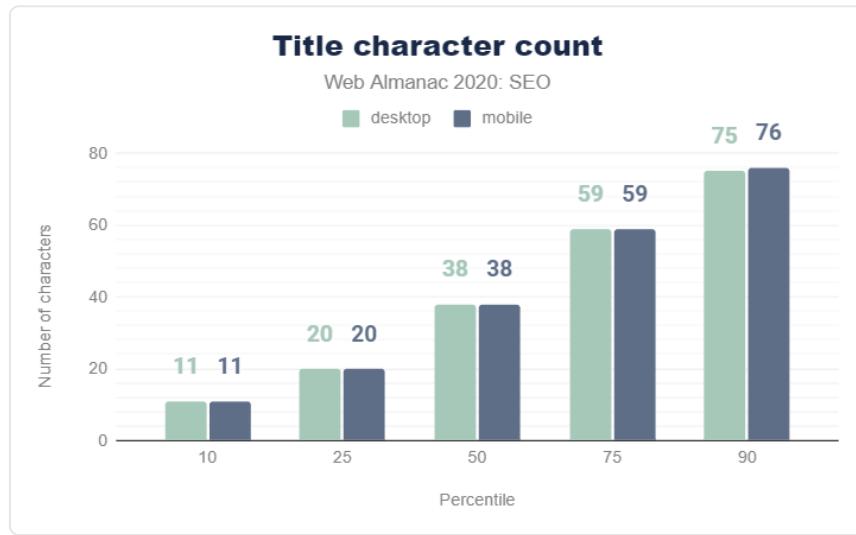


Figure 7.12. Distribution of the number of characters per page title.

Meta descriptions

The meta description acts as the advertising tagline for a web page. Although a recent study suggests that this tag is ignored and rewritten by Google 70% of the time, it is an element that is prepared with the goal of enticing a user to click through.

When analyzing the usage of meta descriptions, we found that 68.62% of desktop pages and 68.22% of mobile pages have one. Although this may be surprisingly low, it is a slight improvement from last year, when only 64.02% of mobile pages had a meta description.

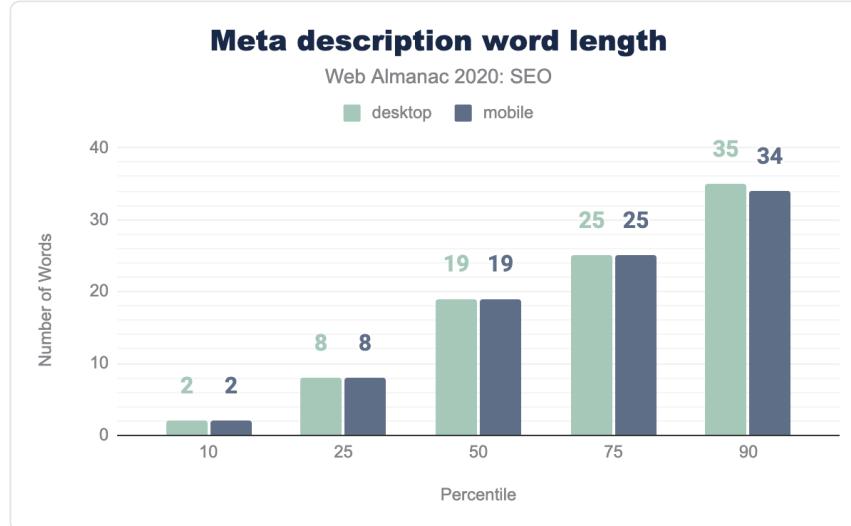


Figure 7.13. Distribution of the number of words per meta description.

The median length of the meta description is 19 words. The only disparity in word count takes place in the 90th percentile where the desktop content has one more word than mobile.

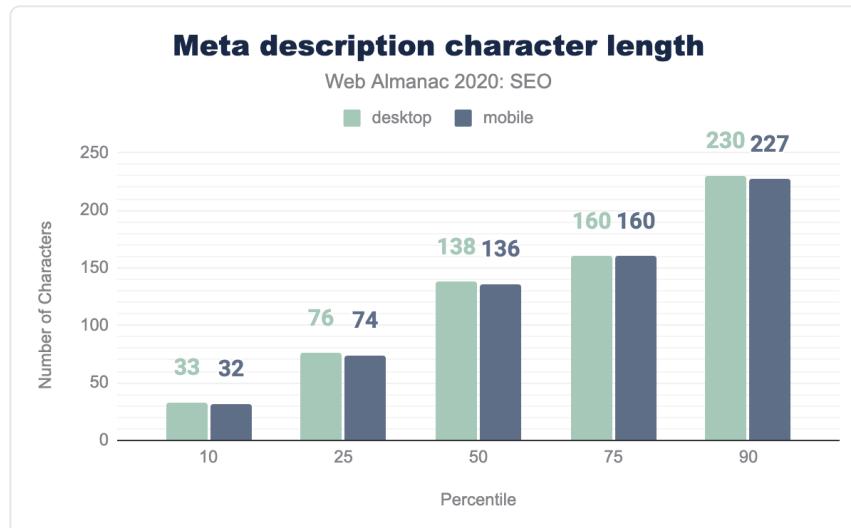


Figure 7.14. Distribution of the number of characters per meta description.

The median character count for the meta description is 138 characters on desktop pages and

136 characters on mobile pages. Aside from the 75th percentile, there is a small disparity between mobile and desktop meta description lengths distributed across the dataset. SEO best practices suggest limiting the specified meta description to up to 160 characters, but Google, inconsistently, may display upwards of 300 characters in its snippets.

With meta descriptions continuing to power other snippets such as social and news feed snippets, and given that Google continually rewrites them and does not consider them a direct ranking factor, it is reasonable to expect that meta descriptions will continue to grow beyond the 160 character limitation.

Images

The usage of images, particularly using `img` tags, within a page often suggests a focus on visual presentation of content. Although search engine capabilities regarding computer vision have continued to improve, we have no indication that this technology is being used in the ranking of pages. `alt` attributes remain the primary way to explain an image in lieu of a search engine's ability to "see" it. `alt` attributes also support accessibility and clarify the elements on the page for users that are visually impaired.

The median desktop page includes 21 `img` tags and the median mobile page has 19 `img` tags. The web continues to trend toward image-heaviness with the growth of bandwidth and the ubiquity of smartphones. However, this comes at a cost of performance.

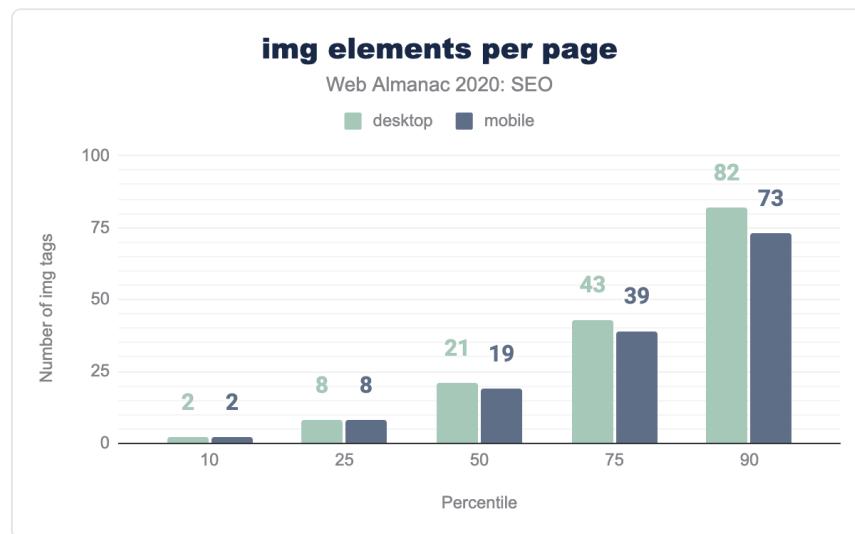


Figure 7.15. Distribution of the number of `` elements per page.

The median web page is missing 2.99% of `alt` attributes on desktop and 2.44% of `alt` attributes on mobile. For more information on the importance of `alt` attributes, see the Accessibility chapter.

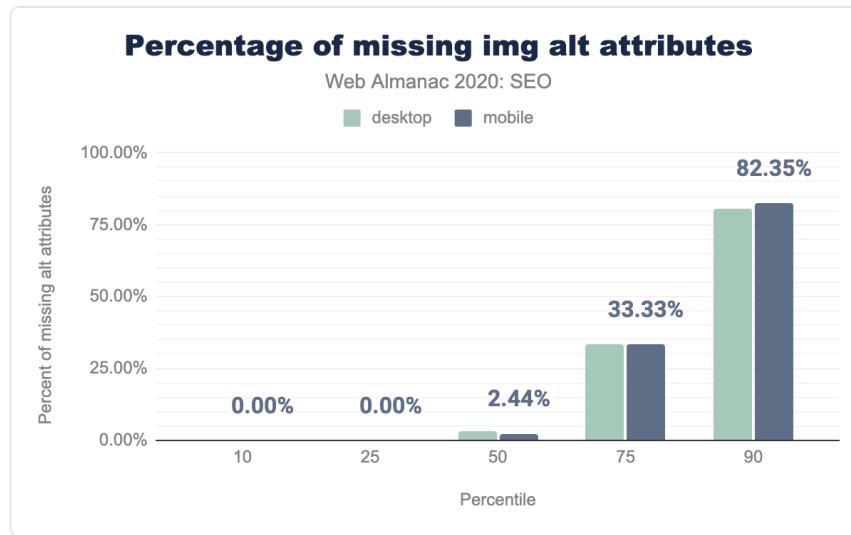


Figure 7.16. Distribution of the percent of `` elements missing image `alt` attributes per page.

We found that the median page contains `alt` attributes on only 51.22% of their images.

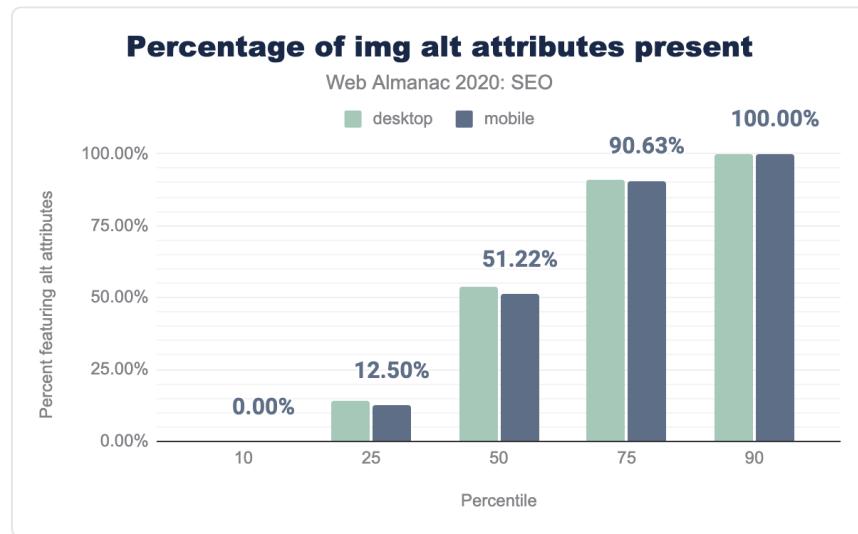


Figure 7.17. Distribution of the percent of images having `alt` attributes per page.

The median web page has 10.00% of images with blank `alt` attributes on desktop and 11.11% on mobile.

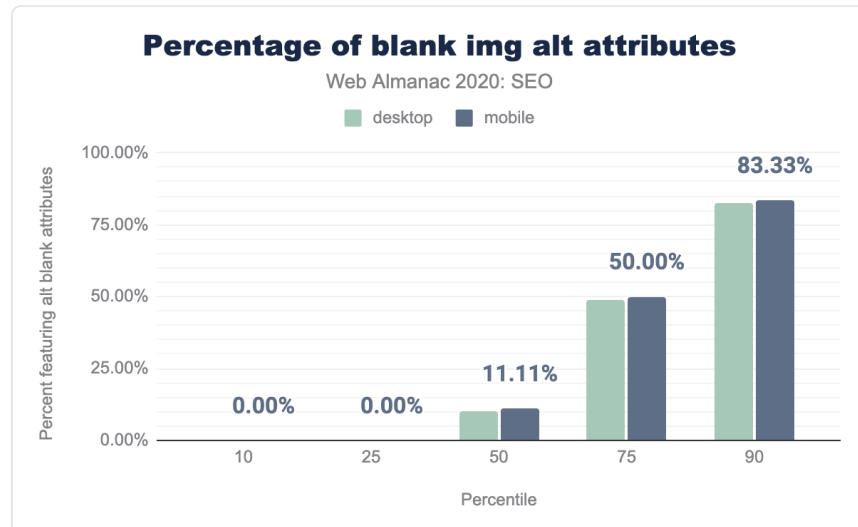


Figure 7.18. Distribution of the percent of images having blank `alt` attributes per page.

Links

Modern search engines use hyperlinks between pages for the discovery of new content for indexing and as an indication of authority for ranking. The link graph is something that search engines actively police both algorithmically and through manual review. Web pages pass link equity through their sites and to other sites through these hyperlinks, therefore it is important to ensure that there is a wealth of links throughout any given page, but also, as Google mentions in its SEO Starter Guide to link wisely.

Outgoing links

As part of this analysis we are able to assess the outgoing links from each page, whether to internal pages from the same domain, as well as external ones, however, have not analyzed incoming links.

The median desktop page includes 76 links while the median mobile page has 67. Historically, the direction from Google suggested that links be limited to 100 per page. While that recommendation is outdated on the modern web and Google has since then mentioned that there are no limits, the median page in our dataset adheres to it.

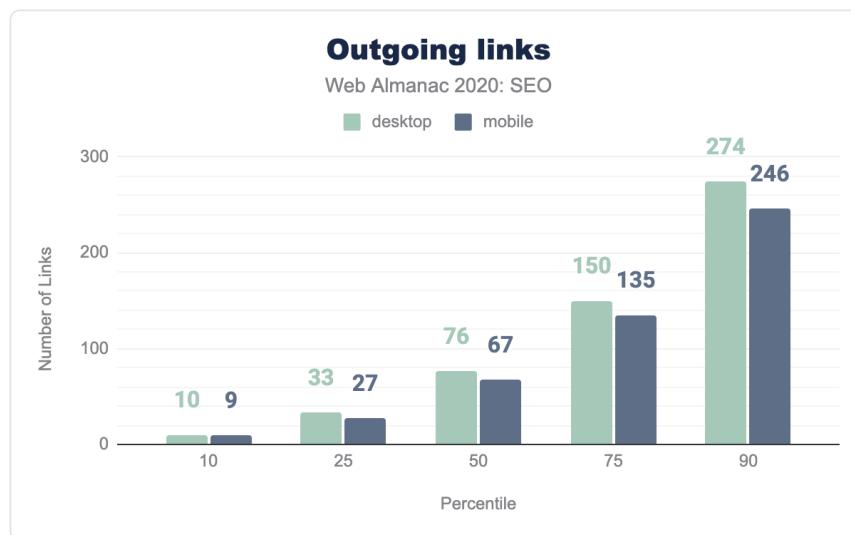


Figure 7.19. Distribution of the number of links per page.

The median page has 61 internal links (going to pages within the same site) on desktop and 54 on mobile. This is down 12.8% and 10% respectively from last year's analysis. This suggests that sites are not maximizing the ability to improve the crawlability and link equity flow through

their pages in the way they did the year before.

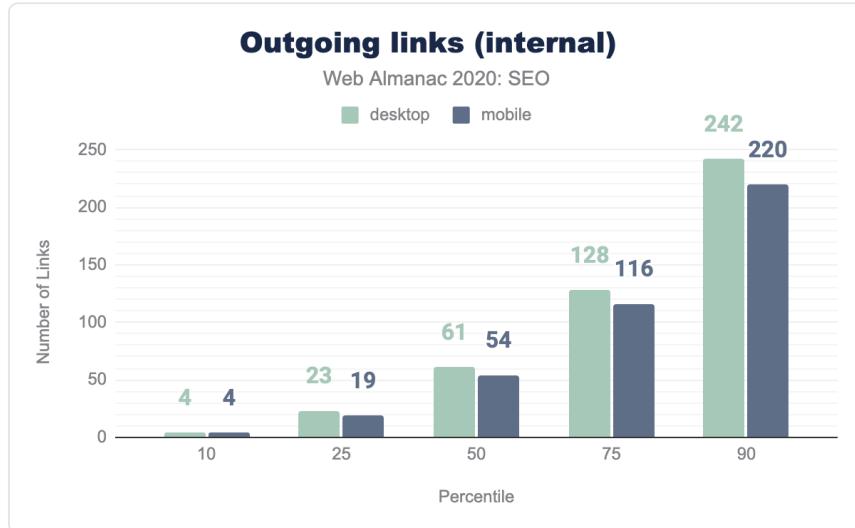


Figure 7.20. Distribution of the number of internal links per page.

The median page is linking to external sites 7 times on desktop and 6 times on mobile. This is a decrease from last year, when it was found that the median number of external links per page were 10 in desktop and 8 on mobile. This decrease in external links could suggest that websites are now being more careful when linking to other sites, whether to avoid passing link popularity or referring users to them.

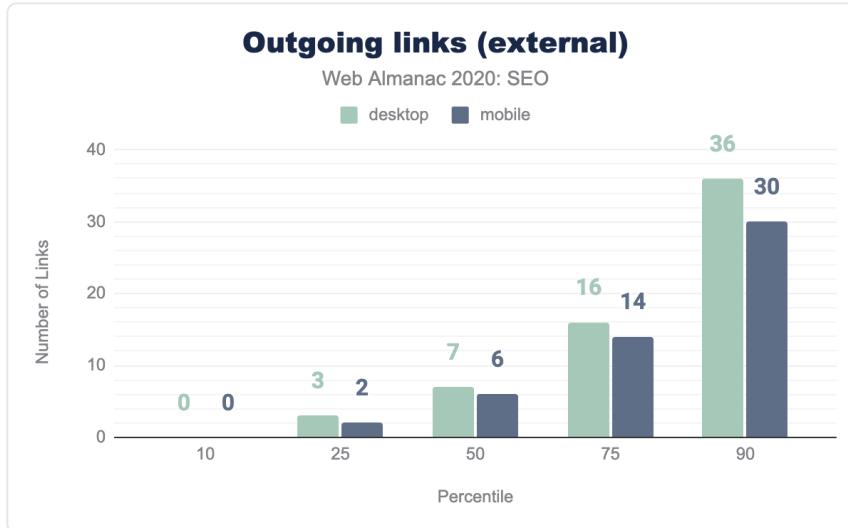


Figure 7.21. Distribution of the number of outgoing external links per page.

Text versus image links

The median web page uses an image as anchor text to link in 9.80% of desktop and 9.82% of mobile pages. These links represent lost opportunities to implement keyword-relevant anchor text. This only becomes a significant issue at the 90th percentile of pages.

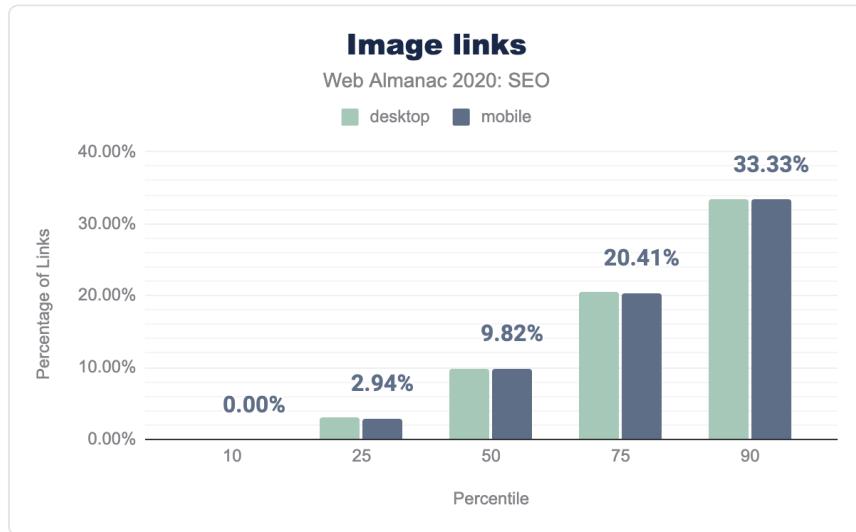


Figure 7.22. Distribution of the percent of links containing images per page.

Mobile versus desktop links

There is a disparity in the links between mobile and desktop that will negatively impact sites as Google becomes more committed to mobile-only indexing rather than just mobile-first indexing. This is illustrated in the 62 links on mobile versus the 68 links on desktop for the median web page.

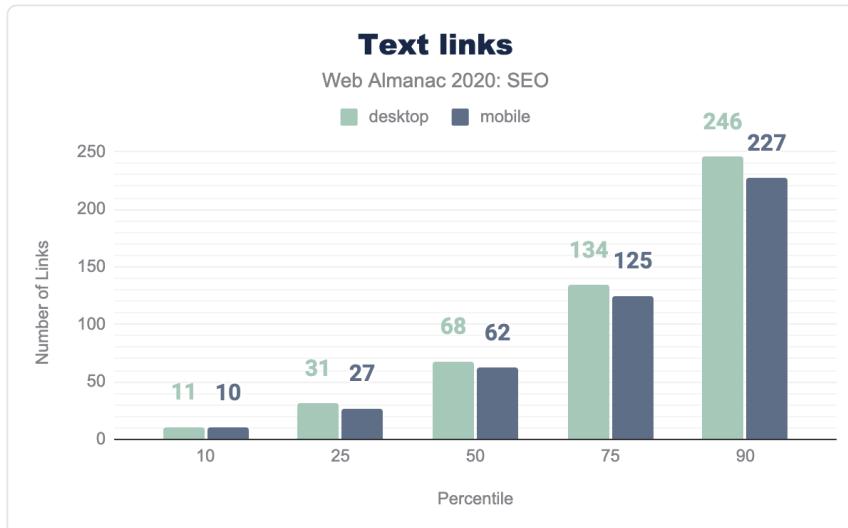


Figure 7.23. Distribution of the number of text links per page.

`rel=nofollow`, `ugc`, and `sponsored` attributes usage

In September of 2019, Google introduced attributes that allow publishers to classify links as being sponsored or user generated content. These attributes are in addition to `rel=nofollow` which was previously introduced in 2005. The new attributes, `rel=ugc` and `rel=sponsored`, are meant to clarify or qualify the reason as to why these links are appearing on a given web page.

Our review of pages indicates that 28.58% of pages include `rel=nofollow` attributes on desktop and 30.74% on mobile. However, `rel=ugc` and `rel=sponsored` adoption is quite low with less than 0.3% of pages (about 20,000) having either. Since these attributes don't add any more value to a publisher than `rel=nofollow`, it is reasonable to expect that the rate of adoption will continue to be slow.

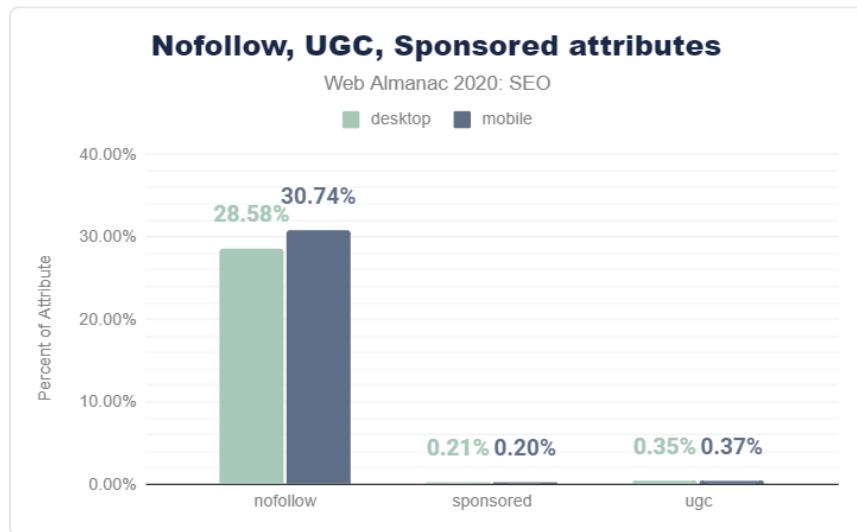


Figure 7.24. Percent of pages having `rel=nofollow`, `rel=ugc`, and `rel=sponsored` attributes.

Advanced

This section explores the opportunities for optimization related to web configurations and elements that may not directly affect a site's crawlability or indexability, but have either been confirmed by search engines to be used as ranking signals or will affect the capacity of websites to leverage search features.

Mobile friendliness

With the increasing popularity of mobile devices to browse and search across the web, search engines have been taking mobile friendliness into consideration as a ranking factor for several years.

Also, as mentioned before, since 2016 Google has been moving to a mobile-first index, meaning that the content that is crawled, indexed, and ranked is the one accessible to mobile users and the Smartphone Googlebot.

Additionally, since July 2019 Google is using the mobile-first index for all new websites and earlier in March, it announced that 70% of pages shown in their search results have already shifted over. It is now expected that Google fully switches to a mobile-first index in March.

2021.

Mobile friendliness should be fundamental for any website looking to provide a good search experience—and as a consequence, grow in search results.

A mobile-friendly website can be implemented through different configurations: by using a responsive web design, with dynamic serving, or via a separate mobile web version. However, maintaining a separate mobile web version is not recommended anymore by Google, who endorse responsive web design instead.

Viewport meta tag

The browser's viewport is the visible area of a page content, which changes depending on the used device. The `<meta name="viewport">` tag (or viewport meta tag) allows you to specify to browsers the width and scaling of the viewport, so that it is correctly sized across different devices. Responsive websites use the viewport meta tag as well as CSS media queries to deliver a mobile friendly experience.

42.98% of mobile pages and 43.2% desktop ones are have a viewport meta tag with the `content=initial-scale=1,width=device-width` attribute. However, 10.84% of mobile pages and 16.18% of desktop ones are not including the tag at all, suggesting that they are not yet mobile friendly.

<i>Viewport</i>	<i>Mobile</i>	<i>Desktop</i>
<code>initial-scale=1,width=device-width</code>	42.98%	43.20%
<code>not-set</code>	10.84%	16.18%
<code>initial-scale=1,maximum-scale=1,width=device-width</code>	5.88%	5.72%
<code>initial-scale=1,maximum-scale=1,user-scalable=no,width=device-width</code>	5.56%	4.81%
<code>initial-scale=1,maximum-scale=1,user-scalable=0,width=device-width</code>	3.93%	3.73%

Figure 7.25. Percent of pages having each viewport meta tag `content` attribute value.

CSS media queries

Media queries are a CSS3 feature that play a fundamental role in responsive web design, as

they allow you to specify conditions to apply styling only when the browser and device match certain rules. This allows you to create different layouts for the same HTML depending on the viewport size.

We found that 80.29% of desktop pages and 82.92% of the mobile ones are using either a `height`, `width`, or `aspect-ratio` CSS feature, meaning that a high percentage of pages have responsive features. The most popularly used features can be seen in the table below.

Feature	Mobile	Desktop
<code>max-width</code>	78.98%	78.33%
<code>min-width</code>	75.04%	73.75%
<code>-webkit-min-device-pixel-ratio</code>	44.63%	38.78%
<code>orientation</code>	33.48%	33.49%
<code>max-device-width</code>	26.23%	28.15%

Figure 7.26. Percent of pages that include each media query feature.

Vary: User-Agent

When implementing a mobile friendly website with a dynamic serving configuration—one in which you show different HTMLs of the same page based on the used device—it is recommended to add a `Vary: User-Agent` HTTP header to help search engines discover the mobile content when crawling the website, as it informs that the response varies depending on the user agent.

Only 13.48% of the mobile pages and 12.6% of the desktop pages were found to specify a `Vary: User-Agent` header.

```
<link rel="alternate" media="only screen and (max-width: 640px)">
```

Desktop websites that have separate mobile versions are recommended to link to them using this tag in the `head` of their HTML. Only 0.64% of the analyzed desktop pages were found to be including the tag with the specified `media` attribute value.

Web performance

Having a fast-loading website is fundamental to provide a great user search experience. Because of its importance, it has been taken into consideration as a ranking factor by search engines for years. Google initially announced using site speed as a ranking factor in 2010, and then in 2018 did the same for mobile searches.

As announced in November 2020, three performance metrics known as Core Web Vitals are on track to be a ranking factor as part of the "page experience" signals in May 2021. Core Web Vitals consist of:

Largest Contentful Paint (LCP)

- Represents: user-perceived loading experience
- Measurement: the point in the page load timeline when the page's largest image or text block is visible within the viewport
- Goal: <2.5 seconds

First Input Delay (FID)

- Represents: responsiveness to user input
- Measurement: the time from when a user first interacts with a page to the time when the browser is actually able to begin processing event handlers in response to that interaction
- Goal: <300 milliseconds

Cumulative Layout Shift (CLS)

- Represents: visual stability
- Measurement: the sum of the number of *layout shift scores* approximating the percent of the viewport that shifted
- Goal: <0.10

Core Web Vitals experiences per device

Desktop continues to be the more performant platform for users despite more users on mobile devices. 33.13% of websites scored *Good* Core Web Vitals for desktop while only 19.96% of their mobile counterparts passed the Core Web Vitals assessment.

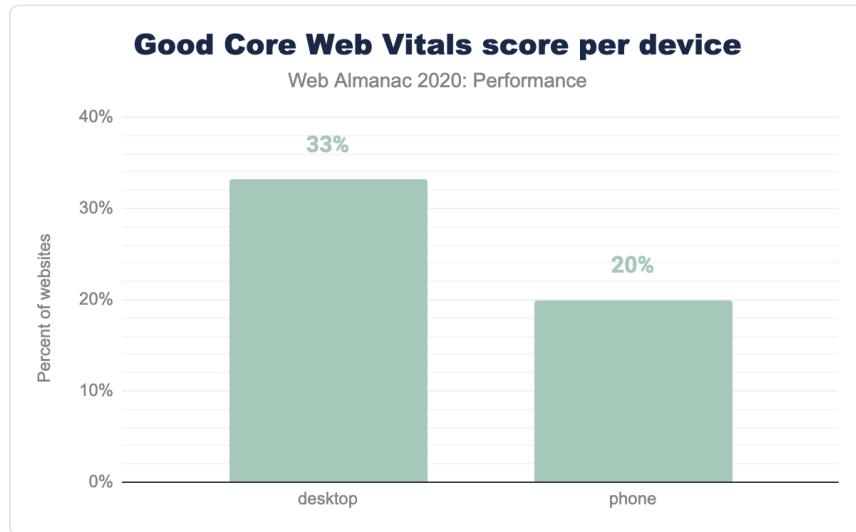


Figure 7.27. Percent of websites passing the Core Web Vitals assessment per device.

Core Web Vitals experiences per country

A user's physical location impacts performance perception as their locally available telecom infrastructure, network bandwidth capacity, and the cost of data create unique loading conditions.

Users located in the United States recorded the largest absolute number of websites with *Good* Core Web Vitals experiences despite only 32% of sites earning the passing grade. Republic of Korea recorded the highest percentage of *Good* Core Web Vital experiences at 52%. The relative portion of total websites requested by each country is worth noting. Users in United States generated 8X the total origin requests as generated by Republic of Korea users.

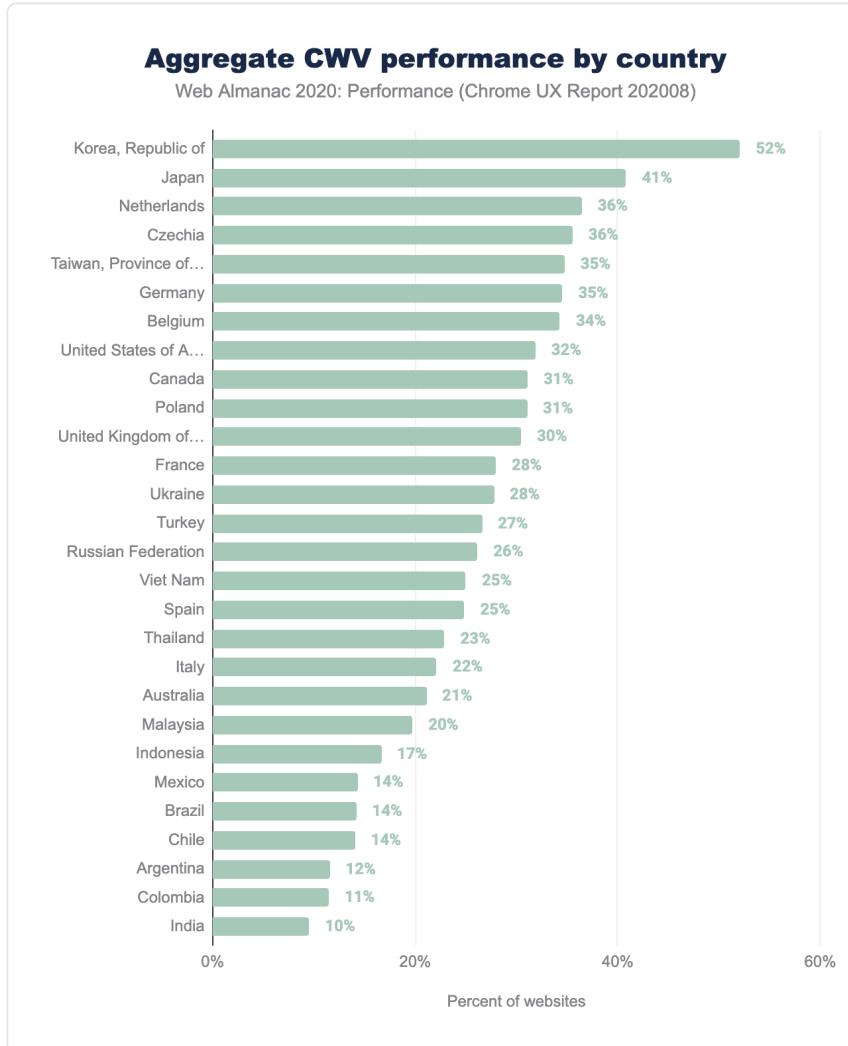


Figure 7.28. Percent of websites passing the Core Web Vitals assessment per country.

Additional analyses of Core Web Vitals performance by dimensions by effective connection type and specific metrics are available in the Performance chapter.

Internationalization

Internationalization covers the configurations that multilingual or multi-country websites can use to inform search engines about their different language and/or country versions, specify

which are the relevant pages to show users in each case, and avoid targeting issues.

The two international configurations that we analyzed are the `content-language` meta tag and the `hreflang` attributes, that can be used to specify the language and the content of each page. Additionally, `hreflang` annotations allow you to specify the alternate language or country versions of each page.

Search engines like Google and Yandex use `hreflang` attributes as a signal to determine the page's language and country target, and although Google doesn't use the HTML lang or the `content-language` meta tag, the latter last tag is used by Bing.

`hreflang`

8.1% of desktop pages and 7.48% of mobile pages use the `hreflang` attribute, which might seem low, but this is natural because these are only used by multilingual or multi-country websites.

We found that only 0.09% of the desktop pages and 0.07% of the mobile pages implement `hreflang` via their HTTP headers, and that most rely on the HTML `head` implementation.

We also identified that there are some pages that rely on JavaScript to render `hreflang` annotations. 0.12% of desktop and mobile pages are showing `hreflang` in the rendered but not in the raw HTML.

From a language and country value perspective, when analyzing the implementation via the HTML `head`, we found that English (`en`) is the most popular used value, with 4.11% of the mobile and 4.64% of the desktop pages including it. After English, the second most popular value is `x-default` (used when defining a *default* or *fallback* version for users of non-targeted languages or countries), with 2.07% of mobile and 2.2% of the desktop pages including it.

The third, fourth and fifth most popular are German (`de`), French (`fr`) and Spanish (`es`), followed by Italian (`it`) and English for the US (`en-us`), as can be seen in the table below with the rest of the values implemented via the HTML `head`.

Value	Mobile	Desktop
<code>en</code>	4.11%	4.64%
<code>x-default</code>	2.07%	2.20%
<code>de</code>	1.76%	1.88%
<code>fr</code>	1.74%	1.87%
<code>es</code>	1.74%	1.84%
<code>it</code>	1.27%	1.33%
<code>en-us</code>	1.15%	1.31%
<code>ru</code>	1.12%	1.13%
<code>en-gb</code>	0.87%	0.98%
<code>pt</code>	0.87%	0.87%
<code>nl</code>	0.83%	0.94%
<code>ja</code>	0.73%	0.81%
<code>pl</code>	0.72%	0.75%
<code>de-de</code>	0.69%	0.78%
<code>tr</code>	0.69%	0.66%

Figure 7.29. Percent of pages that include the top `hreflang` values in the HTML `head`.

Something slightly different was found in top `hreflang` language and country values implemented via the HTTP headers, with English (`en`) being again the most popular one, although in this case followed by French (`fr`), German (`de`), Spanish (`es`) and Dutch (`nl`) as the top values.

Values	Mobile	Desktop
<i>en</i>	0.05%	0.06%
<i>fr</i>	0.02%	0.02%
<i>de</i>	0.01%	0.02%
<i>es</i>	0.01%	0.01%
<i>nl</i>	0.01%	0.01%

Figure 7.30. Percent of pages that include the top `hreflang` values in HTTP headers.

Content-Language

When analyzing the `content-language` usage and values, whether by implementing it as a meta tag in the HTML `head` or in the HTTP headers, we found that only 8.5% of mobile pages and 9.05% of desktop pages were specifying it in the HTTP headers. Even fewer websites were specifying their language or country with the `content-language` tag in the HTML `head`, with only 3.63% of mobile pages and 3.59% of desktop pages featuring the meta tag.

From a language and country value perspective, we found that the most popular values specified in the `content-language` meta-tag and HTTP headers are English (`en`) and English for the US (`en-us`).

In the case of English (`en`) we identified that 4.34% of desktop and 3.69% of mobile pages specified it in the HTTP headers and 0.55% of the desktop and 0.48% of the mobile pages were doing so via the `content-language` meta tag in the HTML `head`.

For English for the US (`en-us`), the second most popular value, it was found that only 1.77% of mobile pages and 1.7% of desktop ones were specifying it in the HTTP headers, and 0.3% of the mobile pages and 0.36% desktop ones were doing it so in the HTML.

The rest of the most popular language and country values can be seen in the tables below.

Value	Mobile	Desktop
<code>en</code>	3.69%	4.34%
<code>en-us</code>	1.77%	1.70%
<code>de</code>	0.50%	0.44%
<code>es</code>	0.34%	0.33%
<code>fr</code>	0.31%	0.34%
<code>ru</code>	0.18%	0.16%
<code>pt-br</code>	0.15%	0.16%
<code>nl</code>	0.13%	0.15%
<code>it</code>	0.13%	0.13%
<code>ja</code>	0.08%	0.10%

Figure 7.31. Percent of pages using the top `content-language` values in HTTP headers.

Value	Mobile	Desktop
<code>en</code>	0.48%	0.55%
<code>en-us</code>	0.30%	0.36%
<code>pt-br</code>	0.24%	0.24%
<code>ja</code>	0.19%	0.26%
<code>fr</code>	0.18%	0.19%
<code>tr</code>	0.17%	0.13%
<code>es</code>	0.16%	0.15%
<code>de</code>	0.15%	0.11%
<code>cs</code>	0.12%	0.12%
<code>pl</code>	0.11%	0.09%

Figure 7.32. Percent of pages using the top `content-language` values in HTML meta tags.

Security

Google places a specific focus on security in all respects. The search engine maintains lists of sites that have shown suspicious activity or have been hacked. Search Console surfaces these issues and Chrome users are presented with warnings before visiting sites with these problems. Additionally, Google provides an algorithmic boost to pages that are served over HTTPS (Hypertext Transfer Protocol Secure). For a more in-depth analysis on this topic, see the Security chapter.

HTTPS usage

We found that 77.44% of desktop pages and 73.22% of mobile pages have adopted HTTPS. This is up 10.38% from last year. It is important to note that browsers have become more aggressive in pushing HTTPS by signaling that pages are insecure when you visit them without HTTPS. Also, HTTPS is currently a requirement to capitalize on higher performing protocols such as HTTP/2 and HTTP/3 (also known as HTTP over QUIC). You can learn more about the state of these protocols in the HTTP/2 chapter.

All of these things have likely contributed to the higher adoption rate year over year.

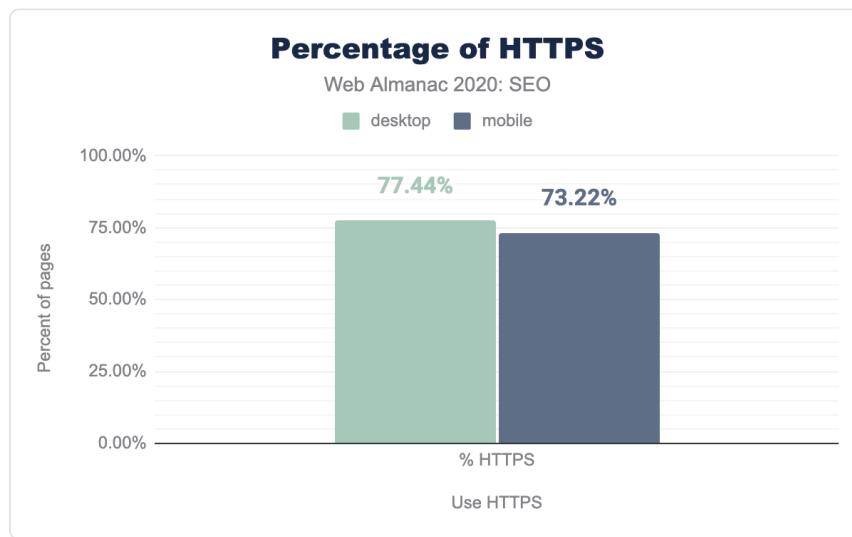


Figure 7.33. Percent of pages served with HTTPS.

AMP

AMP (previously called Accelerated Mobile Pages) is an open source HTML framework that was launched by Google in 2015 as a way to help pages load more quickly, especially on mobile devices. AMP can be implemented as an alternate version of existing web pages or developed from scratch using the AMP framework.

When there's an AMP version available for a page, it will be shown by Google in mobile search results, along with the AMP logo.

It is also important to note that while AMP usage is not a ranking factor for Google (or any other search engine), web speed is a ranking factor.

Additionally, as of this writing, AMP is a requirement to be featured in Google's Top Stories carousel in mobile search results, which is an important feature for news-related publications. However, this will change in May 2021, when non-AMP content will become eligible as long as it meets the Google News content policies and provides a great page experience as announced by Google in November this year.

When checking the usage of AMP as an alternate version of a non-AMP based page, we found that 0.69% of mobile web pages and 0.81% of desktop ones were including an `<amphtml>` tag pointing to an AMP version. Although the adoption is still very low, this is a slight improvement from last year's findings, in which only 0.62% of mobile pages contained a link to an AMP version.

On the other hand, when assessing the usage of AMP as a framework to develop websites, we found that only 0.18% of mobile pages and 0.07% of desktop ones were specifying the `<html amp>` or `<html >>` emoji attribute, which are used to indicate AMP-based pages.

Single-page applications

Single-page applications (SPAs) enable browsers to retain and update a single page load even as the on-page content updates to match a user request. Multiple technologies such as JavaScript frameworks, AJAX, and WebSockets are used to accomplish lightweight subsequent page loads.

These frameworks required special SEO considerations, although Google has worked to mitigate the issues caused by client-side rendering with aggressive caching strategies. In a video from Google Webmaster's 2019 conference, Software Engineer Erik Hendriks shared that Google no longer relies on `Cache-Control` headers and instead looks for `ETag` or `Last-Modified` headers to see if the content of the file has changed.

SPAs should utilize the Fetch API for granular control of caching. The API allows for the passing

of `Request` objects with specific cache overrides set and can be used to set the necessary `If-Modified-Since` and `ETag` headers.

Undiscoverable resources are still the primary concern of search engines and their web crawlers. Search crawlers look for the `href` attributes in `<a>` tags to find linked pages. Without these, the page is seen as isolated without internal linking. 5.59% of desktop pages studied contained no internal links as well as 6.04% of mobile-rendered pages. This is a marker that the page is part of a JavaScript framework SPA and missing the necessary `<a>` tag with valid `href` attributes required for their internal linking to be discovered.

The discoverability of links in popular JavaScript frameworks used for SPAs increased dramatically in 2020 over the previous year. In 2019, 13.08% of mobile navigation links on React sites used deprecated hash URLs. For 2020, only 6.12% of the tested React links were hashed.

Similarly, Vue.js saw a drop to 3.45% from the previous year's 8.15%. Angular was the least likely to use uncrawlable hashed mobile navigation links in 2019 with only 2.37% of mobile pages using them. For 2020, that number plummeted to 0.21%.

Conclusion

Consistent with what was found and concluded last year, most sites have crawlable and indexable desktop and mobile pages, and are making use of the fundamental SEO-related configurations.

It is important to highlight how the link discoverability for major JavaScript frameworks used for SPAs increased dramatically compared to 2019. By testing mobile navigation links for hashed URLs, we saw -53% instances of uncrawlable links from sites using React, -58% fewer from Vue.js powered sites, and a -91% reduction from Angular SPAs.

Additionally, we also identified that there has been a slight improvement from last year's findings across many of the analyzed areas:

- `robots.txt`: Last year 72.16% of mobile sites had a valid `robots.txt` versus 74.91% this year.
- `canonical tag`: Last year we identified that 48.34% of mobile pages were using a canonical tag versus 53.61% this year.
- `title tag`: This year we found that 98.75% of the desktop pages are featuring one, while 98.7% of mobile pages are also including it. Last year's chapter found that 97.1% of mobile pages were featuring a `title` tag.
- `meta description`: This year, we found 68.62% of desktop pages and 68.22% of

mobile ones had a `meta` description, an improvement from last year when 64.02% of mobile pages had one.

- **structured data:** Despite the fact that reviews are not supposed to be associated with home pages, the data indicates that `AggregateRating` is up 23.9% on mobile and 23.7% on desktop.
- **HTTPS usage:** 77.44% of desktop pages and 73.22% of mobile pages have adopted HTTPS. This is up 10.38% from last year.

However, not everything has improved over the last year. The median desktop page includes 61 internal links while the median mobile page has 54. This is down 12.8% and 10% respectively from last year, suggesting that sites are not maximizing the ability to improve the crawlability and link equity flow through their pages.

It is also important to note how there's still an important opportunity for improvement across many critical SEO related areas and configurations. Despite the growing use of mobile devices and Google's move to a mobile-first index:

- 10.84% of mobile pages and 16.18% of desktop ones are not including the `viewport` tag at all, suggesting that they are not yet mobile friendly.
- Non-trivial disparities were found across mobile and desktop pages, like the one between mobile and desktop links, illustrated in the 62 links on mobile versus the 68 links on desktop for the median web page.
- 33.13% of websites scored *Good* Core Web Vitals for desktop while only 19.96% of their mobile counterparts passed the Core Web Vitals assessment, suggesting that desktop continues to be the more performant platform for users.

These findings could negatively impact sites as Google completes its migration to a mobile-first index in March 2021.

Disparities were also found across rendered and non-rendered HTML. For example, the median mobile page displays 11.5% more words when rendered than its raw HTML, indicating a reliance on client-side JavaScript to show content.

Search crawlers look for the `<a href>` tags to find linked pages. Without these, the page is seen as isolated without internal linking. 5.59% of desktop pages contained no internal links as well as 6.04% of mobile-rendered pages.

These findings suggest that search engines are continually evolving in their capacity to effectively crawl, index, and rank websites, and some of the most important SEO configurations are now also better taken into consideration.

However, many sites across the web are still missing out on important search visibility and growth opportunities, which also shows the persisting need of SEO evangelization and best

practices adoption across organizations.

Authors



Aleyda Solis

@aleyda aleysda <https://www.aleydasolis.com/en/>

SEO consultant, author, speaker and entrepreneur. Founder of Orainti²⁹ (a boutique SEO consultancy working with some of the top Web properties and brands, from SaaS to marketplaces) and co-founder of Remoters.net³⁰ (a free remote work hub, featuring remote jobs, tools, events, guides, and more to facilitate remote work). Aleyda enjoys sharing about SEO through her blog³¹, the #SEOFOMO newsletter³², the Crawling Mondays³³ video and podcast series and over Twitter³⁴.



Michael King

@IPullRank ipullrank

An artist and a technologist, all rolled into one, Michael King is the founder of enterprise digital marketing agency, iPullRank³⁵ and founder of Natural Language Generation app CopyScience³⁶. Effortlessly leaning on his background as an independent hip-hop musician, Mr. King is a compelling content creator and an award-winning dynamic speaker who is called upon to contribute to technology and marketing conferences and blogs all over the world. You can find Mike talking trash on Twitter³⁷

29. <https://www.orainti.com/>
 30. <https://remoters.net/>
 31. <https://www.aleydasolis.com/en/blog/>
 32. <https://www.aleydasolis.com/en/seo-tips/>
 33. <https://www.aleydasolis.com/en/crawling-mondays-videos/>
 34. <https://twitter.com/aleyda>
 35. <https://ipullrank.com>
 36. <https://www.copyscience.io>
 37. <https://twitter.com/ipullrank>



Jamie Indigo

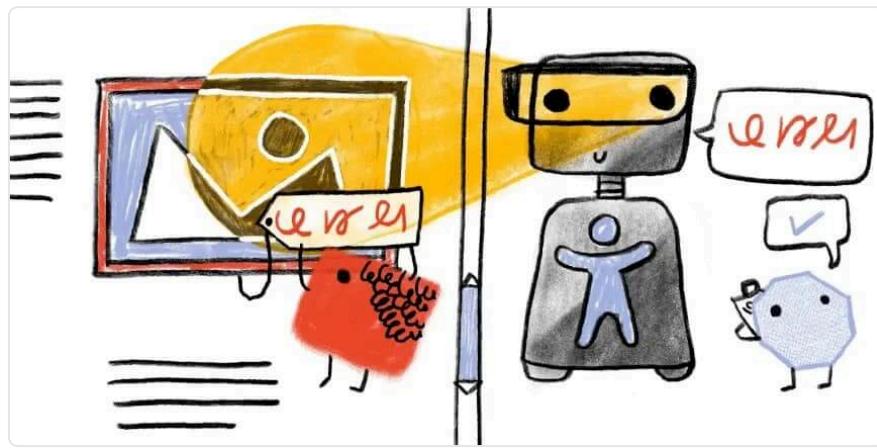
Twitter: [@Jammer_Volts](https://twitter.com/Jammer_Volts) | GitHub: [fellowhuman1101](https://github.com/fellowhuman1101) | Website: <https://not-a-robot.com/>

100% human & totally not a robot, Jamie Indigo untangles technologies to help humans access useful information and businesses provide valuable digital experiences. She founded Not a Robot³⁸ to consult with a focus on the human aspects of technical SEO including ethics & inclusion in technology and the search industry. She can be found learning in public on Twitter³⁹.

38. <https://not-a-robot.com>
39. https://twitter.com/Jammer_Volts

Part II Chapter 8

Accessibility [UNEDITED]



Written by Olu Niyi-Awosusi and Alex Tait

Reviewed by Adrian Roselli, Eric Bailey, and David Fox

Analyzed by David Fox

Introduction

In 2020, more than ever before, it's become increasingly urgent for digital spaces to be inclusive and accessible to all. With the ongoing pandemic making it even more difficult for folks to access services in-person and entire industries moving online, disabled people are disproportionately impacted. Additionally, the number of disabled people is rising due to the effects of the pandemic.

Web accessibility is about achieving feature and information parity and giving complete access to all aspects of an interface to disabled people. A digital product or website is simply not complete if it is not usable by everyone. If it excludes certain disabled populations, this is discrimination and potentially grounds for fines and/or lawsuits.

The Web Content Accessibility Guidelines, or WCAG, is an internationally recognized set of standards that needs to be met in all websites and applications that utilize the Internet. They are not laws, but many laws point to WCAG as their basis.

These guidelines have had multiple releases over the years and the current standard is WCAG 2.1, with WCAG 2.2 currently being vetted as a working draft. Some regional laws point to WCAG 2.0 as the requirement, but as Adrian Roselli covers in his article WCAG 2.1 is the Current Standard, Not WCAG 2.0 - and WCAG 2.2 is Coming we need to be meeting WCAG 2.1 standards and considering the new criteria coming in WCAG 2.2 as well.

A dangerous trend that has seen more exposure than ever in 2020 is the use of “accessibility overlays”. These widgets promise one step accessibility compliance and more often than not introduce new barriers and make the experience for a disabled user quite challenging. It is important that digital practitioners take ownership over designing and implementing usable interfaces and not try to subvert this process with a quick fix. For more information see Lainey Feingold’s article, Honor the ADA: Avoid Web Accessibility Quick Fix Overlays.

Sadly, year over year, we and other teams conducting analysis such as the WebAIM Million are finding little and in some cases no improvement in these metrics. The median overall site score for all lighthouse audit data rose from 73% in 2019 to 80% in 2020. We hope that this 7% increase represents a shift in the right direction. However, these are automated checks and could mean that developers are doing a better job of subverting the rule engine, so we are cautiously optimistic.

Our analysis is based on automated metrics only. It is important to remember that automated testing captures only a fraction of the accessibility barriers that can be present in an interface. Qualitative analysis, including manual testing and usability testing with disabled people are needed in order to achieve an accessible site or application.

We've split up our most interesting insights into five categories:

1. Ease of reading,
2. Media on the web,
3. Ease of page navigation,
4. Assistive technologies on the web,
5. Accessibility of form controls.

Ease of reading

Making content as simple and clear to read as possible is an important aspect of web accessibility. Being unable to read the content of a page prevents a user from being able to complete tasks on websites. There are many aspects of a web page that make it easier or harder to read, including:

Color Contrast

The higher the page contrast, the easier it is for people to view text-based content. People who may have difficulties viewing low contrast content include those with color vision deficiency, people with mild to moderate vision loss, and those with contextual difficulties viewing the content, such as glare on screens in bright light. Unfortunately, only 21.06% of sites were found to have sufficient color contrast. Which is a decrease from last year's already abysmal 22%.

Zooming and Scaling of Pages

The ability to set scale and zoom of desktop pages helps those with vision loss to more easily navigate sites, as well as allowing users of less precise navigation tools easier control of sites.

These were disabled on 24.39% of desktop sites checked, and 30.66% of mobile sites. Some operating systems no longer comply with disabled zoom and scale set in HTML. For systems that do respect it, this can render the page effectively unusable for some. For more information about why to avoid disabling browser zoom see Adrian Roselli's article, [Don't Disable Zoom](#).

Language Identification

Setting an HTML `lang` attribute allows easy translation of a page and better screen reader support. The percentage of sites with a valid HTML lang attribute on desktop this year was 77.67%, with only 77.7% having a `lang` attribute at all.

Media on the web

Images and their text alternatives

Images are an essential part of the web experience. They can add an enriched context to the surrounding textual information, and not just for sighted users. In 1995, HTML 2.0 introduced the `alt` attribute, enabling web authors to provide a text alternative for the visual information communicated in an image. A screen reader can convey its visual meaning aurally by announcing the image's alternative text. Additionally if images are unable to load, the alternative text for a description will be displayed.

The 2020 Lighthouse audit data shows that only 54% of sites pass the test for images with `alt` text. This test looks for the presence of at least one of the `alt`, `aria-label` and `aria-labelledby` attributes on `img` elements. In most cases using the `alt` attribute is the best choice. Even though `alt` attributes have been around for 25 years, we also found that 21.24%

of desktop images and 21.38% of mobile images are lacking alternative text. This is one of the easiest automated checks to test for using your accessibility tool of choice, and should be low hanging fruit and a relatively straightforward problem to solve.

Screen reader users listen to the “aural UI” as described by Steve Faulker, an aural or sonic experience of the interface wherein the structure, semantics and relationships of the content are announced. This means that screen reader users consume a lot of textual information. For this reason it is important to assess whether or not an image might not need to be described. This is a helpful decision tree from the W3C for deciding how and whether to describe an image. If an image is truly decorative and adds nothing meaningful to the surrounding context, you can assign the `alt` attribute a null value, `alt=""`. It is important to do this explicitly rather than omitting the `alt` attribute altogether, as omitting it could lead to assistive technology announcing the image path, which is a very confusing user experience. We found that 26.20% of desktop pages and 26.23% of mobile pages contain `alt` attributes with a null/empty value. We hope this indicates that over a quarter of websites are being developed with consideration for which images are truly meaningful and not as a means of side-stepping automated checks.

When describing an image it is imperative to consider what information the user needs, and omit additional information to reduce verbosity. For example, a red arrow icon button that has the action of moving to a new step in the interface could be described as “continue to step 3 of 5” rather than “red arrow png”. The first description tells the user what to expect if they activate the control, whereas the second just describes its appearance and has an unnecessary file extension, both of which are irrelevant to the meaning of the image.

Automated checks for the presence of alternative text do not assess the quality of this text. As described in the previous section, the meaning of an image needs to be considered when writing this text. One common unhelpful pattern is describing the image with the file extension name. For the previous “red arrow png” example, a screen reader user likely does not get helpful information from the image format. We found that 6.8% of desktop sites (with at least one instance of the `alt` attribute) had a file extension in its value.

The top 5 file extensions explicitly included in the `alt` text value (for sites with images that have non-empty alt values) are `jpg`, `png`, `ico`, `gif`, and `jpeg`. This likely comes from a CMS or another auto-generated alternative text mechanism. It is imperative that these alt attribute values be meaningful, regardless of how they are implemented.

File extension type	Desktop	Mobile
<i>jpg</i>	3.730%	3.500%
<i>png</i>	2.980%	2.810%
<i>ico</i>	1.340%	1.600%
<i>gif</i>	0.034%	0.030%
<i>jpeg</i>	0.034%	0.032%

Figure 8.1. Percent of pages with non-empty alt attributes.

Images with title attributes

The `title` attribute which generates a tooltip that displays text is often mistaken as another reliable way to describe images to assistive technology. However according the HTML Standard,

"Relying on the `title` attribute is currently discouraged as many user agents do not expose the attribute in an accessible manner as required by this specification"

Tooltips also introduce a host of other accessibility barriers such as information only being revealed on hover/mouseover, information not being properly communicated to assistive technology, lack of keyboard support, and general poor usability. The history of tooltips and their barriers are well described by Sarah Higley in her blog post, "Tooltips in the time of WCAG 2.1". We found that 16.95% of all alt attributes also contain a title attribute and of these instances 73.56% of the titles are the exact same as the alt attribute. Of these instances 73.56% of desktop sites and 72.80% of mobile sites had matching values for both the alt and title attributes.

Other facts about alt text

15,357,625

Figure 8.2. The longest known alt text length.

- The median length for both desktop and mobile `alt` text is 18 characters. With the average English word length being 4.7 characters, this means the median alt attribute value is 3-4 words long. Depending on the image, being terse can be

beneficial. However it is hard to imagine 4 words being sufficient for an accurate description of an image with any complexity.

- The longest `alt` text length found for desktop sites was 15,357,625 characters. That's enough to fill 5 and a half "War and Peace" sized books (assuming "War and Peace" has an average word length of 4.7 characters).

Video on the Web

Video and other multi-media content can enrich a Web experience, but often is not robustly supported for all users and can pose major accessibility barriers if it is not implemented with support. For more information see the W3C's Making Audio and Video Accessible.

Captions

Captions or transcripts are needed to communicate aural information for people who are deaf or hard of hearing, and are very also helpful for users who have cognitive disabilities such as audio processing difficulty. Transcripts also help low-vision and blind users by describing visuals. Video content on the web is not accessible if it does not have accompanying captions. Similar to the importance of having meaningful alternative text for images, the quality of captions is also very important.

"Captions not only include dialogue, but identify who is speaking and include non-speech information conveyed through sound, including meaningful sound effects"

-WCAG, Understanding Success Criterion 1.2.2: Captions

Of sites using `<video>` elements, only 0.79% provide closed captions, which we assume based on the presence of the `<track>` element (and which are different from open / burned-in captions). Note that some websites have custom solutions for providing video and audio captions to users. We were unable to detect these custom solutions so the percentage of sites utilizing captions could be higher, but this figure is indicative of how under supported captions are on Web video content. We also cannot assess the quality of the captions detected and whether or not they accurately convey the full meaning of the video they describe.

Autoplaying video

It is arguably a disruptive and undesirable user experience to autoplay and loop video on a website for all users. Video can be a resource drain for device batteries as well as data, and in some cases video can contain content that is distressing for users, whether by showing disturbing imagery or being used as an attack vector against people prone to seizures.

For disabled users there are significant barriers caused by autoplaying and looping video. For screen reader users, a video that contains audio will likely disrupt the announcements and lead to confusion. For folks with cognitive disabilities such as ADHD video can be very distracting and interrupt the user's ability to use and understand the interface. People with vestibular conditions can be dangerously triggered by video as well.

The Web Content Accessibility Guidelines has a criteria 2.2.2 Pause, Stop, Hide that requires that any moving, blinking or scrolling content (including video) that plays for longer than 5 seconds have a mechanism to pause, stop, or hide it.

Of the pages with video present, we found that 56.98% of desktop pages and 53.64% of mobile pages have the `autoplay` attribute, meaning that videos play by default. We also found that 58.42% of desktop pages and 52.86% of mobile pages have the `loop` attribute, which very likely means the video plays indefinitely. Though there could be mechanisms to pause, stop or hide these videos, opting into playing video rather than needing to stop autoplaying and/or looping the video should be the default. These metrics suggest that over half of websites with video could have significant accessibility barriers.

Ease of page navigation

Pages need to be easy to navigate so users are not left feeling lost, or unable to find the content they need to do what brought them to our sites in the first place. Screen reader technology also needs to be able to differentiate between different sections, so users of this software are not left with an indecipherable wall of text.

Headings

Headings make it easier for screen readers to properly navigate a page by supplying a hierarchy that can be jumped through like a table of contents.

Our audits revealed that 58.72% of the sites checked pass the test for properly ordered headings that do not skip levels. These headings convey the semantic structure of the page. Many screen reader users navigate a page through its headings, so having them in the correct order - ascending with no jumps - means that assistive technology users will have the best experience. It is worth noting that we only check pages where these rules are more likely to be followed; home pages are more likely than interior pages to follow this rule.

Skip links

Skip links enable a user to skip through any interactive content such as a navigation system and go to another destination, typically the main content of the page. They are typically the first link on a page and can be persistent in the UI or visibly hidden until they have keyboard focus. This prevents keyboard users from needing to potentially tab through an extraneous number of elements to get to the content they are trying to access.

Skip links are considered a bypass for a block. The 2020 Lighthouse audit data revealed that 93.90% of sites pass the bypass block test, meaning they have a `<header>`, skip link or landmark region to allow users to skip repetitive content.

Tables

Tables are an efficient way to display data with two axes of relationships, making them useful for comparisons. Users of assistive technology rely on specific accessibility features designed to navigate properly structured tables in order to have the best experience navigating and interacting with them. Without valid semantic table markup present, these features cannot be used.

Table captions

Table captions act as a label for the table supplying context for the table data. Only 4.98% of desktop sites and 4.20% of mobile sites with a table used a table caption.

Presentational tables

We are fortunate in 2020 to have so many flexible CSS methodologies that allow for fluid responsive layouts. However, many years ago before the likes of Flexbox and CSS Grid, developers often used tables for layout. Unfortunately due to some combination of legacy websites and legacy development techniques there are still sites out there where tables are used for layout.

If there is an absolute need to reach for this technique, the role of presentation should be applied to the table such that assistive technology will ignore the table semantics. We found that 0.063% of desktop and 0.049% of mobile pages had a table with a role of presentation. It's hard to know if this is good or bad. It could indicate that there are not many tables used for presentational purposes, but it is very likely that tables used for layout are just lacking this needed role.

Document titles

Descriptive page titles are helpful for context when moving between pages, tabs and windows with assistive technology because the change in context will be announced. Our data shows 98.98% of sites have a title which is a hopeful statistic. However it stands to reason that home pages may have a higher rate of page titles than less important routes.

Tabindex

Tabindex dictates the order in which focus moves throughout the page. Interactive content such buttons, links and form controls have a natural `tabindex` value of `0`. Similarly, custom elements and widgets that are intended to be interactive and in the keyboard focus order need an explicitly assigned `tabindex="0"`. If a non-interactive element should be focusable but not in the keyboard tab order a `tabindex` value of `-1` can be used allowing for focus to be programmatically set with JavaScript.

The focus order of the page should always be determined by the document flow. Setting the `tabindex` to a positive integer value overrides the natural order of the page and is considered bad practice. Respecting the natural order of the page generally leads to a more accessible experience. We found that 5% of desktop sites and 4.34% of mobile sites used positive integers as tab index values.

Assistive technologies on the Web

People with varying disabilities use different assistive technologies to help them experience the Web. This Tools and Techniques article from the Web Accessibility Initiative or WAI of the W3C covers how users can perceive, understand and interact with the Web using different assistive technologies.

Some assistive technologies for the Web include:

- Screen readers
- Voice control
- Screen magnifiers
- Input devices (such as pointers and switch devices)

Screen readers present content audibly, usually by the computer speaking or announcing the content in the interface as the user navigates and interacts. This enables blind, low vision, and other disabled and non-disabled users to consume the content without needing to rely on the visual cues displayed on the screen.

ARIA

In 2014 the WAI published Accessible Rich Internet Applications, or ARIA. They describe ARIA as, "WAI-ARIA, the Accessible Rich Internet Applications Suite, defines a way to make Web content and Web applications more accessible to people with disabilities. It especially helps with dynamic content and advanced user interface controls developed with Ajax, HTML, JavaScript, and related technologies."

Most developers think of ARIA as attributes we can add to HTML to make it more usable for screen reader users, but it was never intended to make up for improper markup and native HTML solutions. ARIA has a lot of nuances which, when misunderstood, can introduce new accessibility barriers. Furthermore different screen readers have varying limitations with respect to ARIA support.

The Five Rules of ARIA

There are 5 rules of ARIA that we need to understand before making use of this powerful toolset. This is not an official specification with required conformance, but a guide for understanding and implementing ARIA correctly.

The 5 Rules of ARIA (from W3C's Using Aria):

1. If you can use a native HTML element [HTML 5.1] or attribute with the semantics and behavior you require already built in, instead of repurposing an element and adding an ARIA role, state or property to make it accessible, then do so.
2. Do not change native semantics, unless you really have to.
3. All interactive ARIA controls must be usable with the keyboard.
4. Do not use `role="presentation"` or `aria-hidden="true"` on a focusable element.
5. All interactive elements must have an accessible name.

ARIA roles

One of the most common ways that ARIA is used is by explicitly defining the role for an element, which communicates its purpose to assistive technology.

HTML 5 introduced many new native elements, all which have implicit semantics, including roles. For example the `<nav>` element has an implicit `role="navigation"` and does not need to have this role added explicitly in order to convey its purpose information to assistive technology. Currently 64.54% of desktop pages have at least one instance of an ARIA role

attribute. The median site has 2 instances of the `role` attribute.

Just use a button!

We found that 25.20% of desktop sites and 24.50% of mobile sites had homepages with at least one element with an explicitly assigned `role="button"`. This suggests that about a quarter of websites are using the `button` role on elements in order to change their semantics, with the exception of buttons that have been explicitly assigned the button role, which is redundant but harmless.

If non-interactive elements such as `<div>`s and ``s have been given this role, there is a significant chance one or more of the 5 rules of ARIA have been broken.

It is fairly likely that a native `<button>` element would be a better choice, per the first rule of ARIA. It is also possible that the role has been added but the expected keyboard support has not been supplied, which would break the 3rd rule of ARIA and would violate WCAG 2.1.1, Keyboard. We found that 8.27% of desktop pages and 8.28% of mobile pages had at least one occurrence of a `<div>` or a `` element with `role="button"` explicitly defined. This act of adding ARIA roles, or a "role-up", is less ideal than using the correct native HTML element.

We found that 15.50% of desktop pages and 14.62% of mobile pages contained at least one anchor element with `role="button"`. If a role has been applied to an element that should have its implicit role respected, such as giving a `role="button"` to a link (which has an implicit `role="link"`), this would break the 2nd rule of ARIA and would violate WCAG 2.1.1, Keyboard if the correct keyboard behavior has not been implemented (links are not activated with the space key, where as buttons are).

In the vast majority of these cases, a better pattern than explicitly defining `role="button"` on the element in question would be to leverage the native HTML `<button>` element.

Navigation

We found that 22.06% of desktop pages and 21.76% of mobile pages have at least one element with `role="navigation"`, which is a landmark role. Per the 1st rule of ARIA, rather than adding this role to an element, developers should be leveraging the HTML 5 `<nav>` element which comes with the correct semantics implicitly. It is possible that this role has been added explicitly to the `<nav>` element, which would not be an accessibility issue, though it is redundant.

Dialog Modals

There are many potential accessibility barriers associated with dialog modals. We recommend reading Scott O'Hara's article [Having an Open Dialog](#) for more context.

We are pleased to report that 19.01% of desktop pages and 18.21% of mobile pages have at least one occurrence of `role="dialog"` which is up from about 8% in 2019. It is worth noting some of the increase is probably due to changes in how this metric was measured. This could also suggest that more developers are considering accessibility when building dialogs and potentially that frameworks and associated packages may be implementing more accessible dialog patterns as well. However, making a dialog modal accessible requires a lot more than using the `dialog` role. Focus management, proper keyboard support, and screen reader exposure all need to be addressed.

Tabs

Tabs are a common interface widget, but present a challenge for many developers to make accessible. A common pattern for accessible implementation comes from the WAI-ARIA Authoring Practices Design Patterns. Please note that the ARIA Authoring Practices document is not a specification, and is meant to show idealized ARIA constructs. They should not be used in production without testing with your users.

In this pattern, a parent container has a `role="tablist"` with children elements that have a `role="tab"`. These tabs are associated with elements that have a `role="tabpanel"`, and contain the content for that tab.

Example

Nils Frahm Agnes Obel Joke

Nils Frahm is a German musician, composer and record producer based in Berlin. He is known for combining classical and electronic music and for an unconventional approach to the piano in which he mixes a grand piano, upright piano, Roland Juno-60, Rhodes piano, drum machine, and Moog Taurus.

Figure 8.3. Tab list (`role="tablist"`) contains all of the tabs. (Source: W3C)

Example

Nils Frahm

Agnes Obel

Joke

Nils Frahm is a German musician, composer and record producer based in Berlin. He is known for combining classical and electronic music and for an unconventional approach to the piano in which he mixes a grand piano, upright piano, Roland Juno-60, Rhodes piano, drum machine, and Moog Taurus.

Figure 8.4. "Nils Frahm" tab (`role="tab"`). (Source: W3C)

Example

Nils Frahm is a German musician, composer and record producer based in Berlin. He is known for combining classical and electronic music and for an unconventional approach to the piano in which he mixes a grand piano, upright piano, Roland Juno-60, Rhodes piano, drum machine, and Moog Taurus.

Figure 8.5. Tab panel (`role="tabpanel"`) with content associated with the "Nils Frahm" tab.
(Source: W3C)

For desktop pages, 7.00% have at least one element with a `role="tablist"` whereas there only 5.79% of pages have elements with a `role="tab"` and 5.46% of pages have elements with a `role="tabpanel"`. This suggests that the pattern may only be partially implemented. Even if there is dynamic rendering at play for some of the tab/tabpanel elements, the currently visible or first tab/tabpanel would theoretically be in the DOM on page load.

Presentation

When an element has been given a `role="presentation"` its semantics are stripped away, for both the element it is assigned to and its required children. For example, tables and lists both have required children, so if the parent has a `role="presentation"` this essentially cascades to the child elements, which will also have their semantics stripped. Removing an element's semantics means that it is no longer that element in any capacity except for its visual appearance. For example, a list with a `role="presentation"` will no longer communicate any information to a screen reader about the list structure.

A common usage of this attribute is for `<table>` elements that have been used for layout

rather than for tabular data. We do not recommend using tables in this way. For layout, we have powerful CSS tools today such as Flexbox and CSS Grid. In general there are very few use cases where `role="presentation"` is particularly helpful for assistive technology users, use this role sparingly and thoughtfully.

Role	Desktop	Mobile
<code>button</code>	25.20%	24.51%
<code>navigation</code>	22.97%	21.76%
<code>dialog</code>	19.01%	18.22%
<code>search</code>	17.94%	17.59%
<code>presentation</code>	17.83%	16.31%

Figure 8.6. Top 5 ARIA roles on the web.

ARIA Attributes

ARIA attributes can be assigned to HTML elements to enhance the accessibility of the interface. Respecting the first rule of ARIA, they should not be used to achieve something that can be done with native HTML.

Labeling and describing elements with ARIA

The browser's accessibility tree has a computation system that assigns the accessible name (if there is one) to a control, widget, group, or landmark such that it can be announced by assistive technology. There is a specificity ranking that happens to determine which value is assigned to the accessible name.

The accessible name can be derived from an element's content (such as button text), an `attribute` (such as an image `alt` text value), or an associated element (such as a programmatically associated label for a form control). For more information about accessible names see Léonie Watson's article, [What is an accessible name?](#)

We can also use ARIA to provide accessible names for elements. There are 2 ARIA attributes that accomplish this, `aria-label`, `aria-labelledby`. Either of these attributes will "win" the accessible name computation and override the natively derived accessible name, so use them with caution and be sure to test with a screen reader or look at the accessibility tree to confirm that the accessible name is what was expected. When using ARIA to name an element, it is

important to ensure that the WCAG 2.5.3, Label in Name criterion has not been violated, which expects visible labels to be at least a part of its accessible name.

The `aria-label` element allows a developer to provide a string value and this will be used for the accessible name for the element. We found that 40.44% of desktop pages and 38.72% of mobile home pages had at least one element with the `aria-label` attribute, making it the most popular ARIA attribute for providing accessible names.

The `aria-labelledby` attribute accepts an `id` reference as its value, which associates it with another element in the interface to provide its accessible name. The element becomes “labelled by” this other element which supplies its accessible name. We found that 17.73% of desktop pages and 16.21% of mobile pages had at least one element with the `aria-labelledby` attribute.

Again, the first rule of ARIA should be respected. If the element can derive its accessible name without needing ARIA, this is preferable. For example a `<button>`, which is not a graphical element, should get its accessible name from its text content rather than an ARIA attribute. Form elements should derive their accessible names from properly associated `<label>` elements whenever possible.

The `aria-describedby` attribute can be used in cases where a more robust description is needed for an element. It also accepts an `id` reference as its value to connect with descriptive text that exists elsewhere in the interface. It does not supply the accessible name, it should be used in conjunction with an accessible name as a supplement, not a replacement. We found that 11.31% of desktop pages and 10.56% of mobile pages had at least one element with the `aria-describedby` attribute.

Fun fact!

We found 3200 websites with the attribute `aria-labeledby`, which is a misspelling of the `aria-labelledby` attribute! Be sure to run those automated checks to pick up these easily avoidable errors.

Hiding content

There are several ways to ensure that assistive technology will not discover content. We can leverage CSS `display:none`; or `visibility:hidden`; to omit the elements from the accessibility tree. If an author wishes to hide content from screen readers specifically they can use `aria-hidden="true"`. We found that 48.09% of desktop pages and 48.23% of mobile pages had at least one instance of an element with the `aria-hidden` attribute.

These techniques are particularly helpful when something in the visual interface is redundant

or unhelpful to assistive technology users. It should be used thoughtfully as it is essential to deliver feature parity for all users. Avoid using it to skip over content that is challenging to make accessible.

Hiding and showing content is a prevalent pattern in modern interfaces, and it can be helpful to declutter the UI for everyone. There are two ARIA attributes that are helpful additions to this disclosure pattern. The `aria-expanded` attribute should have a `true / false` value that toggles depending on whether the disclosed content is shown or not. Additionally the `aria-controls` attribute can be associated with an `id` on the disclosed content creating a programmatic relationship between the triggering control (which should be a button) and the content that gets displayed.

We found that 20.98% of desktop pages and 21.00% of mobile pages had at least one element with the `aria-expanded` attribute and 17.38% of desktop pages and 16.94% of mobile pages had at least one element with the `aria-controls` attribute. This suggests that around 1/5th of websites might be implementing at least partially accessible disclosure widgets. Note that the `aria-controls` attribute is considered a best practice for the disclosure pattern because screen reader support is not ideal.

Screen reader only text

A common technique that developers often employ to supply additional information for screen reader users is to use CSS to visually hide a passage of text such that it will be announced by a screen reader, but not visually present in the interface. Since `display:none` and `visibility:hidden` both prevent content from being present in the accessibility tree, there is a common “hack” involving a chunk of CSS code that will accomplish this. The most common CSS class names for this code snippet (both by convention and throughout libraries like bootstrap) are ‘`sr-only`’ and ‘`visually-hidden`’. We found that 13.31% of desktop pages and 12.37% of mobile pages had one or both of these CSS class names.

Announcing Dynamically Rendered Content

One of the biggest accessibility challenges in modern web development is handling dynamically rendered content which is everywhere in interfaces. The presence of new or updated things in the DOM often needs to be communicated to screen readers. Some thought needs to be put into which updates need to be conveyed. For example, form validation errors need to be conveyed where as a lazy loaded image may not. There also needs to be done in a way that is not disruptive to a task in progress.

One tool we have to help with this is ARIA live regions. Live regions allow us to listen for changes in the DOM, such that the updated content can be announced by a screen reader.

Typically the `aria-live` attribute is placed on its own container element that is already present in the DOM rather than an element that is dynamically rendered. It is important to determine a dedicated node in the DOM that has no chance of being dynamically manipulated by other factors for the live region, ensuring that the announcements are reliable. When elements within this container dynamically render or update (for example, status updates or notification that a form was not successfully submitted) the changes will be announced.

We found that 16.84% of desktop pages and 15.67% of mobile pages have live regions. This attribute has three potential values: `polite`, `assertive`, and `off`. Typically the `polite` value is used, partly because it is the default value, but also because the announcement of the dynamic content will only happen once the user stops interacting with the page. In many cases this is the desired user experience, rather than interrupting their input. If a status update is critical enough, use `assertive` and it will disrupt the screen reader's current speech queue. If it is set to `off` the announcement will not happen. It is important that the natural screen reader experience and flow be respected and that the `assertive` announcements be reserved for extreme cases, and not used for things like marketing announcements.

Disabling browser zoom

It is essential that we allow users to zoom the page or content. There are techniques that can be used to try to disable the ability to scale or zoom the browser. Some operating systems subvert this harmful pattern, but many do not and it is an anti-pattern that needs to be avoided.

Zooming is particularly useful for users with low vision. According to the World Health Organization, "Globally, 1 billion people have a vision impairment". We found that 29.34% of desktop pages and 30.66% of mobile pages attempt to disable scaling by setting either `maximum-scale` to a value less than 1, or `user-scalable` `0` or `none`.

Accessibility of Form Controls

Forms are one of the most important things to get right in terms of accessibility. Successful submission of form input means users can perform core operations of websites and applications. For example if a registration flow is inaccessible, a disabled user might never be able to access the site at all. It is important to remember that digital accessibility is a civil right and that all people have an equal right to access information and perform the same functions on the Web. If a disabled user is prevented from executing core Web tasks or accessing information, especially for tasks like submitting forms for government services and other essential activities, there is a clear-cut case for discrimination in both private and public sectors.

Form validation

It is very important that any form error handling be communicated to assistive technology. There are a variety of techniques for handling this depending on the validation implementation. Web AIM's Usable and Accessible Form Validation and Error Recovery article is a great resource for learning more about various accessible form validation strategies. If a form element is required this also needs to be communicated to assistive technology. For native HTML form elements the required attribute can be used and for customized elements the aria-required attribute may be needed. If there is an issue with a form submission, this needs to be conveyed to assistive technology.

Form labels

Form labels should be visible and persistent in the UI and descriptive of the input they are asking for. It's a good idea to put unique requirements such as formatting or special characters in the visible label so that errors can be prevented whenever possible.

It is important to ensure that form labels have a programmatic association with their respective inputs. It is not sufficient to just display the label visually. We found that only 26.51% of sites have all of their labels properly associated with their respective inputs (achieved with a `for / id` relationship or inputs nested inside labels).

Groups of form controls such as a set of radio inputs or checkboxes should be nested within a `<fieldset>` element and given a group label via the `<legend>` element within the `<fieldset>`. The individual controls still need to be programmatically associated with their respective visible labels as well.

Placeholder text

Do not rely on placeholder text to act as the label for an input. While some screen readers now have the capability of determining the accessible name from placeholder text, users with cognitive disabilities can be negatively impacted by a reliance on placeholder text because as soon as a user begins to type in the input the placeholder disappears and the context is gone. Voice control users need more than a placeholder value in order to reliably target an element in the DOM. Additionally placeholder text often fails color contrast requirements, which negatively impacts users with low vision.

Of the sites that have form controls with placeholder text, 73.89% of them have at least one instance where there is no `<label>` element programmatically associated with the control for desktop and 74.52% for mobile.

Conclusion

This chapter is fittingly included in the User Experience section of this Almanac. As accessibility advocate Billy Gregory once said, “when UX doesn’t consider ALL users, shouldn’t it be known as SOME User Experience, or SUX”. Too often accessibility work is seen as an addition, an edge case, or even comparable to technical debt and not core to the success of a website or product as it should be.

Accessibility is not the sole responsibility of developers to implement. The entire product team and organization have to have it as part of their accountabilities in order to succeed.

Accessibility work needs to shift left in the product cycle, meaning it needs to be baked into the research, ideation and design stages before it is developed.

Potential accessibility responsibilities by role

This list is not exhaustive and is intended to encourage thought about how all of the roles can work together to achieve accessible websites and applications, like a relay race of accountability.

- Human Resources/People Ops:
 - Recruiting and hiring people with accessibility skills including disabled practitioners.
 - Creating an inclusive work environment where people’s disabilities are accommodated.
- UX /Product designers:
 - Considering and talking to people with a range of disabilities in the research and ideation stages.
 - Annotating wireframes with accessibility information such as intended heading hierarchy, skip links, alternative text suggestions (which could also come from copywriters/content folks) and screen reader only text.
- UI designers:
 - Color contrast choices, font selections, spacing and line height considerations.
 - Animation considerations (determining if they are necessary, supplying static assets for `prefers-reduced-motion` scenarios, designing pause/stop mechanisms).
- Product managers:
 - Prioritizing accessibility work in the roadmap, ensuring it does not become technical debt at the end of a backlog.
 - Creating processes for teams to validate their work such as including accessibility in the definition of done and acceptance criteria.

- Developers:
 - Preferring native HTML solutions whenever possible, understanding ARIA and when to use it.
 - Validating all work with automated and manual testing, evaluating colleagues' pull requests with the same criteria.
- Quality Assurance
 - Including accessibility testing in their workflow.
 - Advocating for accessibility considerations when contributing to the team's quality strategy and acceptance criteria.
- Leadership/C-Suite
 - Giving employees bandwidth to learn and grow their accessibility skillset and hiring practitioners with expertise and lived experiences.
 - Considering accessibility core to the product outcomes and viewing accessibility excellence as promotable work.

The tech industry needs to move towards inclusion-driven development. Although this requires some up-front investment, it is much easier and likely less expensive over time to build accessibility into the entire cycle such that it can be baked into the product rather than trying to retrofit sites and apps that were constructed without it in mind.

The largest investment should come in the form of education and process improvements. Once a UI designer understands the nuances of color contrast requirements, selecting an accessible color palette should be the same effort as an inaccessible palette. Once a developer deeply understands native HTML and ARIA and when to reach for certain techniques and tools, the amount of code they write should be comparable.

As an industry it's time that we acknowledge the story told by the numbers in this chapter; we are failing disabled people. We need to do better, and this has to come from a combination of top-down leadership and investment and bottom-up effort to push our practices forward and advocate for the needs, safety and inclusion of disabled people using the Web.

Authors



Olu Niyi-Awosusi

[oluoluoxenfree](https://github.com/oluoluoxenfree) <https://www.opentagclosetag.com/>

Olu Niyi-Awosusi is a software engineer at the FT who loves lists, learning new things, Bee and Puppycat, social justice, accessibility⁴⁰ and trying harder every day.

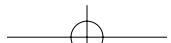
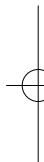
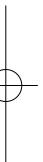
40. <https://alistapart.com/article/building-the-woke-web/>



Alex Tait

🐦 @at_fresh_dev Ⓛ alextait1 ⌂ <https://togethertech.ca/>

Alex Tait is a developer, consultant and educator whose passion lies in the intersection of accessibility and modern JavaScript within interface architecture and design systems. As a developer, she believes that inclusion driven development practices with accessibility at the forefront lead to better products for everyone. As a consultant and strategist, she believes that less is more and that new feature scope creep cannot be prioritized over core feature parity for disabled users. As an educator, she believes in removing barriers to information so that tech can become a more diverse, equitable and inclusive industry.



Part II Chapter 9

Performance



Written by Karolina Szczur

Reviewed by Boris Schapira, Rick Viscomi, David Fox, Noam Rosenthal, Leonardo Zizzamia, and Shane Exterkamp

Analyzed by Max Ostapenko and Pokidov N. Dmitry

Introduction

There is an unquestionable, detrimental effect that slow speed has on overall user experience, and consequently, conversions. But poor performance doesn't just cause frustration or negatively affects business goals—it creates real-life barriers to entry. This year's global pandemic made those existing barriers even more apparent. With the shift to remote learning, work and socializing, suddenly our entire lives were moved online. Poor connectivity and lack of access to capable devices made this change painful at best, if not impossible, to many. It has been a real test, highlighting connectivity, device and speed inequalities worldwide.

Performance tooling continues to evolve to portray those diverse aspects of user experience and make it easier to find underlying issues. Since last year's Performance chapter, there have been numerous significant developments in the space that have already transformed how we approach speed monitoring.

With a significant release of the popular quality auditing tool, Lighthouse 6, the algorithm behind the famous Performance Score has changed significantly, and so did the scores. Core Web Vitals, a set of new metrics portraying different aspects of user experience, has been released. It will be one of the factors for search ranking in the future, shifting the eyes of the development community onto the new speed signals.

In this chapter, we will be looking at real-world performance data provided by the Chrome User Experience Report (CrUX) through the lens of those new developments as well as analyzing a handful of other relevant metrics. It is important to note that due to the limitations of iOS, CrUX mobile results don't include devices running Apple's mobile operating system. This fact will undeniably affect our analysis, especially when examining metric performance on a per-country basis.

Let's dive in.

Lighthouse Performance Score

In May 2020, Lighthouse 6 was released. The new major version of the popular performance auditing suite introduced notable changes to its Performance Score algorithm. The Performance Score is a high-level portrayal of site speed. In Lighthouse 6, the score is measured with six—not five—metrics: First Meaningful Paint and First CPU Idle were removed and replaced with Largest Contentful Paint (LCP), Total Blocking Time (TBT, the lab equivalent of First Input Delay) and Cumulative Layout Shift (CLS).

The new scoring algorithm is prioritizing the new generation of performance metrics: Core Web Vitals and deprioritizing First Contentful Paint (FCP), Time to Interactive (TTI) and Speed Index, as their score weight decreases. The algorithm also now distinctly emphasizes three aspects of user experience: *interactivity* (Total Blocking Time and Time to Interactive), *visual stability* (Cumulative Layout Shift) and *perceived loading* (First Contentful Paint, Speed Index, Largest Contentful Paint).

Additionally, the score is now calculated using different reference points for desktop and mobile. What this means in practice is that it will be less forgiving on desktop (expecting fast websites) and more relaxed on mobile (since benchmark performance on mobile is less quick than desktop). You can compare your Lighthouse 5 and 6 score difference in the Lighthouse Score calculator. So, how did the scores really change?

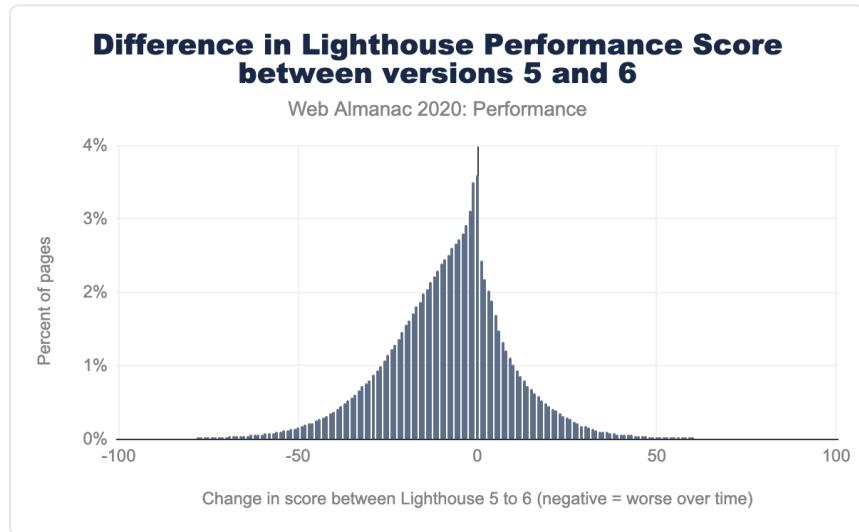


Figure 9.1. Difference in Lighthouse Performance Score between versions 5 and 6.

Above, we observe that 4% of websites recorded no Performance Score change, but the number of sites with negative changes outweighs the ones with score improvements. The Performance Score grades have gotten worse (with the most prominent decrease curve in the 10-25 points area), which is portrayed even more directly below:



Figure 9.2. Lighthouse Performance Score distribution for Lighthouse 5 and 6.

In the comparison of Lighthouse 5 and Lighthouse 6, we can directly observe how the distribution of score has changed. With the Lighthouse 6 algorithm, we observe a rise in the percentage of pages receiving scores between 0 to 25 and a decline between 50 and a 100. While in Lighthouse 5, we saw 3% of pages receiving 100 scores, on Lighthouse 6, that number fell to only 1%.

These overall changes are not unexpected considering a multitude of amendments to the algorithm itself.

Core Web Vitals: Largest Contentful Paint

Largest Contentful Paint (LCP) is a landmark timing-based metric that reports the time at which the largest above-the-fold element was rendered.

LCP by device

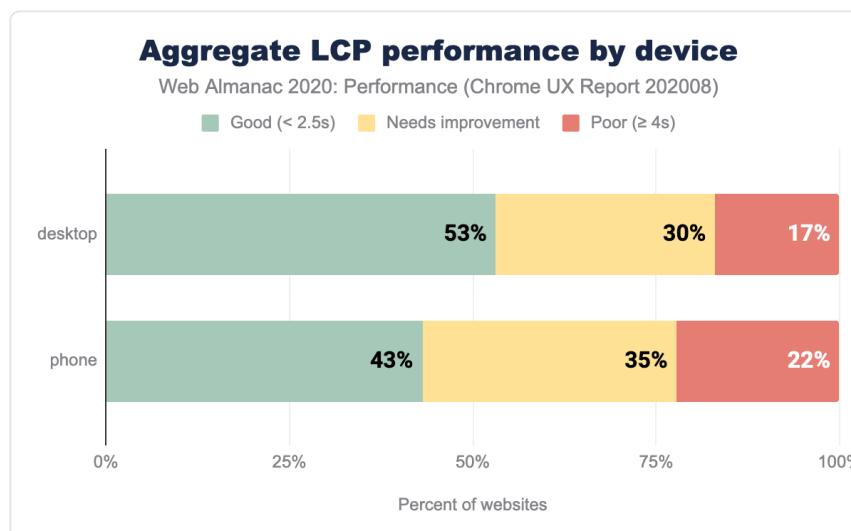


Figure 9.3. Aggregate LCP performance split by device type.

In the chart above, we can observe that between 43% and 53% of websites have good LCP performance (below 2.5s): the majority of websites manage to prioritize and load their critical, above-the-fold media fast. For a relatively new metric, this is an optimistic signal. The slight variance between desktop and mobile is likely to be caused by varying network speeds, device capabilities and image sizing (large, desktop-specific images will take longer to be downloaded

and rendered).

LCP by geographic location

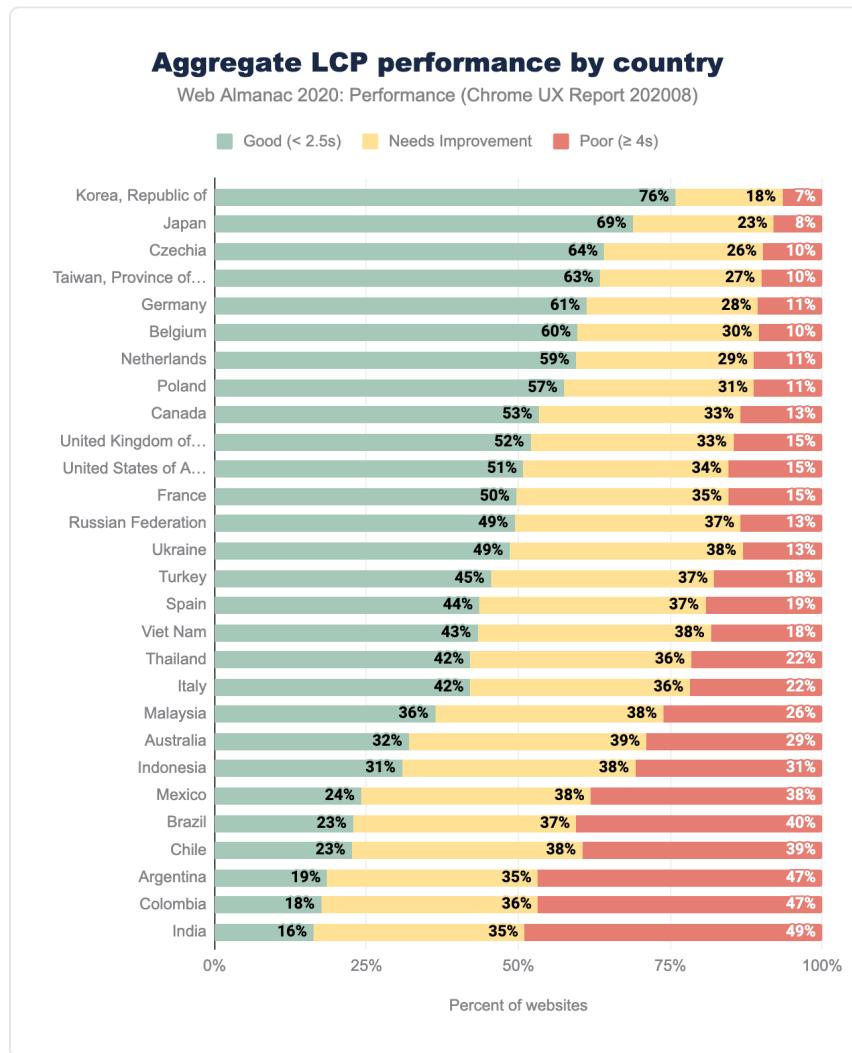


Figure 9.4. Aggregate LCP performance split by country.

The highest percentage of good LCP readings is mostly distributed amongst European and Asian countries with the Republic of Korea (South Korea) leading at 76% of good metric readings. South Korea is a consistent leader in mobile speeds, with an impressive 145 Mbps

download reported by Speedtest Global Index for October. Japan, Czechia, Taiwan, Germany and Belgium are also a handful of countries with reliably high mobile speeds. Australia, while leading in mobile network speeds, is let down by slow desktop connections and latency which places it at the bottom section of the ranking above.

India remains the last one in our set of data, with only 16% of good experiences. While the population of people connecting to the internet for the first time is continually growing, the mobile and desktop network speeds are still an issue, with average downloads of 10Mbps for 4G, 3Mbps for 3G and below 50Mbps for desktop.

LCP by connection type

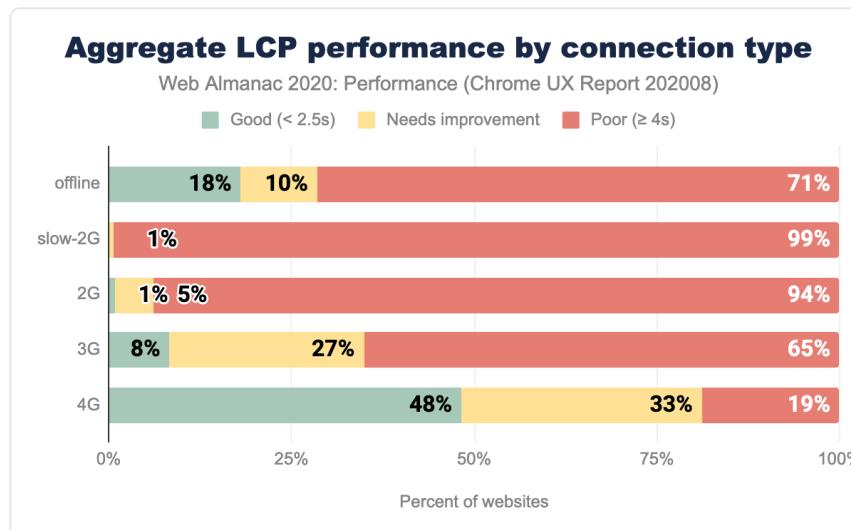


Figure 9.5. Aggregate LCP performance split by connection type.

Since LCP is a timing showcasing when the largest above-the-fold element has been rendered (including imagery, videos or block-level elements containing text), it is not surprising that the slower the network, the more significant portion of measurements are poor.

There's a clear correlation of network speed and better LCP performance, but even on 4G, only 48% of results are categorized as good, leaving half of the readings in need of an improvement. Automating media optimization, serving the right dimensions and formats, as well as optimizing for Low Data Mode, could help move the needle. Learn more about optimizing LCP in this guide.

Core Web Vitals: Cumulative Layout Shift

Cumulative Layout Shift (CLS) quantifies how much elements within the viewport move around during the page visit. It helps pinpoint the degree to which unexpected movement occurs on your websites to grade the user experience, rather than attempting to measure a specific part of interaction with the help of a unit like seconds or milliseconds.

In that way, CLS is a different, new type of a UX holistic metric in comparison to others mentioned in this chapter.

CLS by device

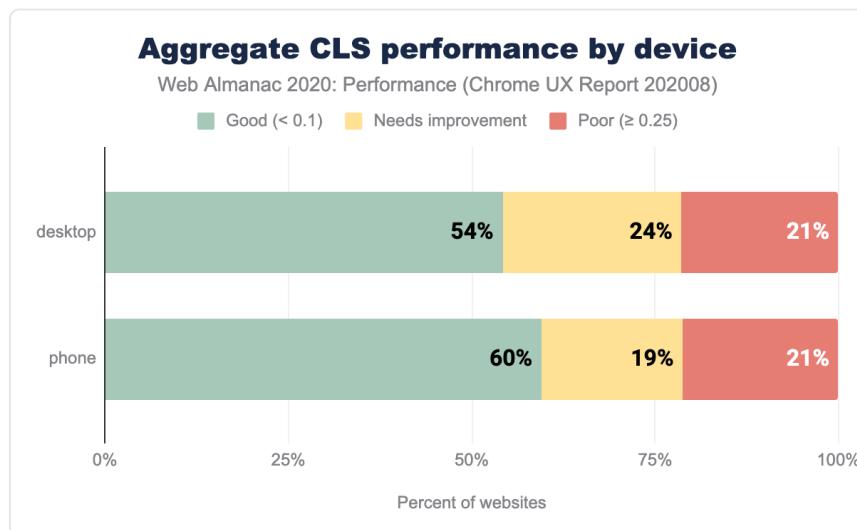


Figure 9.6. Aggregate CLS performance split by device type.

According to CrUX data, both in cases of desktop and mobile devices, more than half of the websites have a good CLS score. There's a slight difference (6 percentage points) between the number of good-rated websites between desktop and mobile, favoring the latter. We could speculate that phones lead in good CLS ratings due to mobile-optimized experiences that tend to be less feature and visuals-rich.

CLS by geographic location

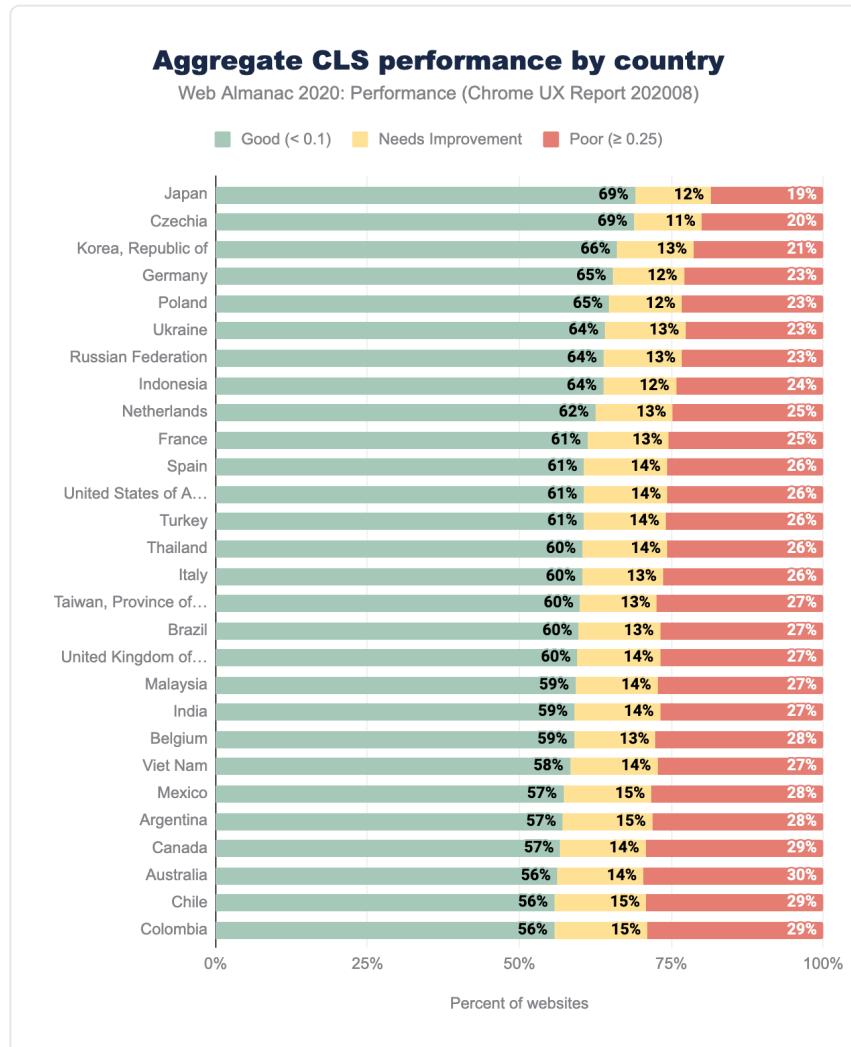


Figure 9.7. Aggregate CLS performance split by country.

The CLS performance in different geographical regions is primarily good, with at least 56% of sites with a good rating. This is excellent news for perceived visual stability. We can observe similar countries leading as we've seen in the LCP geo-distribution—Republic of Korea, Japan, Czechia, Germany, Poland.

Visual stability is less affected by geography and latency to other metrics, like LCP. The difference in the percentage of good scores between the top and the bottom country is 61% for LCP and only 13% for CLS. The percentage of moderate-rating websites is relatively low across the board, giving way to 19%-29% of poor experiences across the board. There are numerous factors that can contribute to poor CLS—learn how to address them in the Optimize Cumulative Layout Shift guide.

CLS by connection type

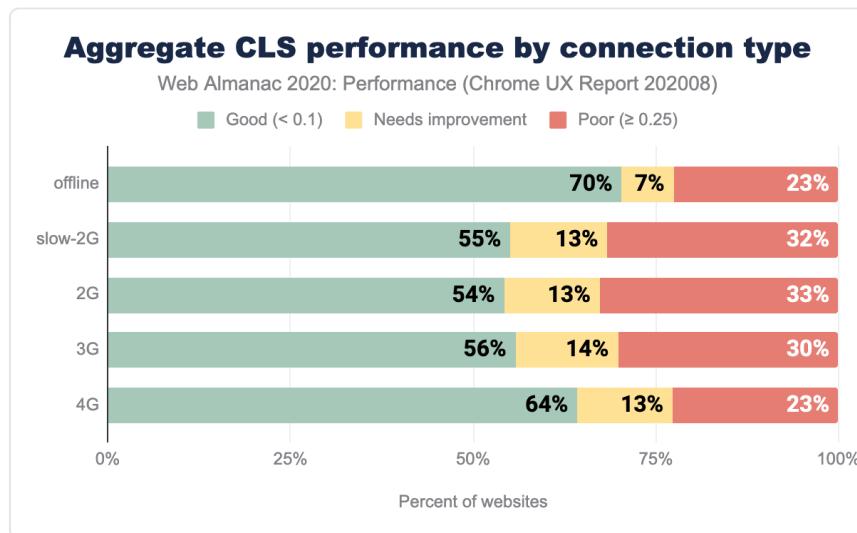


Figure 9.8. Aggregate CLS performance split by connection type.

There's a reasonably even distribution of CLS scoring across most connection types except for offline and 4G. In the offline scenario, we can speculate that Service Workers serve websites. Consequently, there's no delay in download caused by connection type, resulting in the most significant portion of good grades.

It is challenging to draw definite conclusions about 4G, but we can speculate that perhaps 4G+ connections are used as a method of internet access on desktop devices. If that assumption was valid, web fonts and imagery could be heavily cached, which positively affects CLS measurements.

Core Web Vitals: First Input Delay

First Input Delay (FID) measures the time between first user interaction and when the browser is able to respond to that interaction. FID is a good indicator of how interactive your websites are.

FID by device

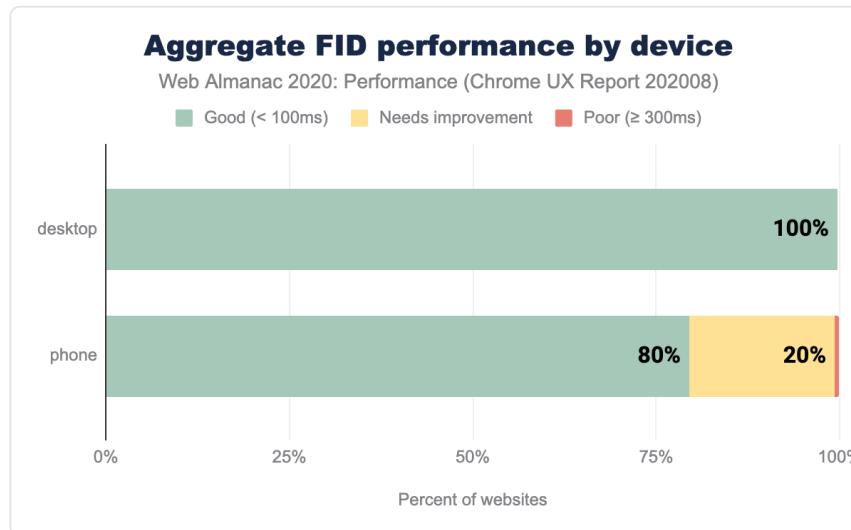


Figure 9.9. Aggregate FID performance split by device type.

It is relatively uncommon to see good scores distributed across such a high percentage of websites. On desktop, based on the 75th percentile of sites' distributions, 100% of sites report fast timings for FID, meaning the number of people experiencing interaction delays is very low.

On mobile, 80% of sites are graded as good. A likely explanation is the reduced CPU capabilities in comparison to desktop, network latency on mobile (causing a delay in script download and execution) as well as battery efficiency and temperature limitations, capping the CPU potential of mobile devices. All of which directly affect interactivity metrics like FID.

FID by geographic location

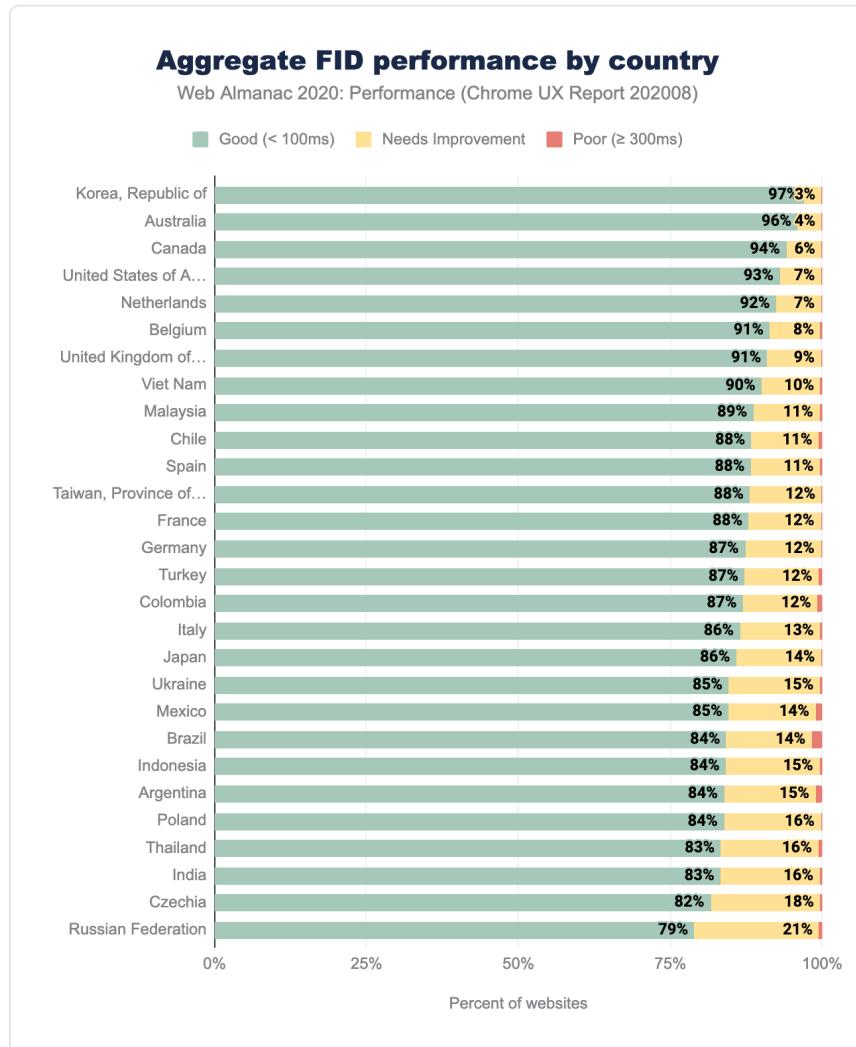


Figure 9.10. Aggregate FID performance split by country.

The geographic distribution of FID scoring confirms the findings in the aggregate device chart shared earlier on. At worst, 79% of websites have good FID, with an impressive 97% on the top position with the Republic of Korea leading again. Interestingly, some top contenders from the CLS and LCP ranking, such as Czechia, Poland, Ukraine and Russian Federation here fall to the bottom of the hierarchy.

Again, we can speculate why that might be, but would need further analysis to really be sure. Assuming FID correlates to JavaScript execution capabilities, countries where more capable devices are more expensive and treated as luxury items, might report lower FID ranking. Poland is a good example—with one of the highest iPhone prices compared to the US market, combined with relatively lower wages, a single salary isn't sufficient to purchase Apple's flagship product. To contrast, Australians with average salaries would be able to buy an iPhone with a weeks' worth of pay. Luckily, the percentage of websites with a low rating is mostly at 0, with a few exceptions of 1-2% readings, pointing towards a relatively speedy response to the interaction.

FID by connection type

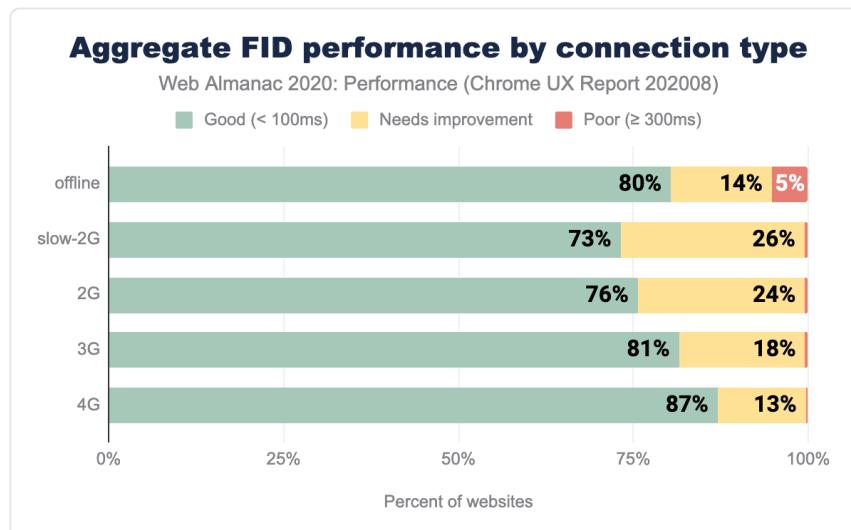


Figure 9.11. Aggregate FID performance split by connection type.

We can observe a direct correlation between network speed and fast FID, ranging from 73% on 2G to 87% on 4G networks. Faster networks will aid in speedier script download, which consequently speeds up the beginning of the parsing and fewer tasks blocking the main thread. It is optimistic to see such results, especially when the ratio of poorly rated site experiences doesn't exceed 5%.

First Contentful Paint

First Contentful Paint (FCP) measures the first time at which the browser rendered any text, image, non-white canvas or SVG content. FCP is a good indicator of perceived speed as it

portrays how long people wait to see the first signs of a site loading.

FCP by device

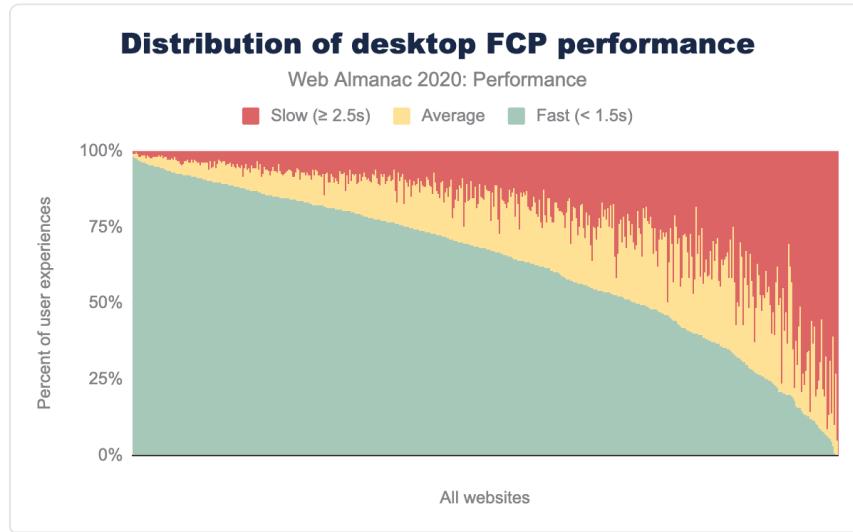


Figure 9.12. Distribution of websites labeled as having fast, average and slow FCP performance on desktop.

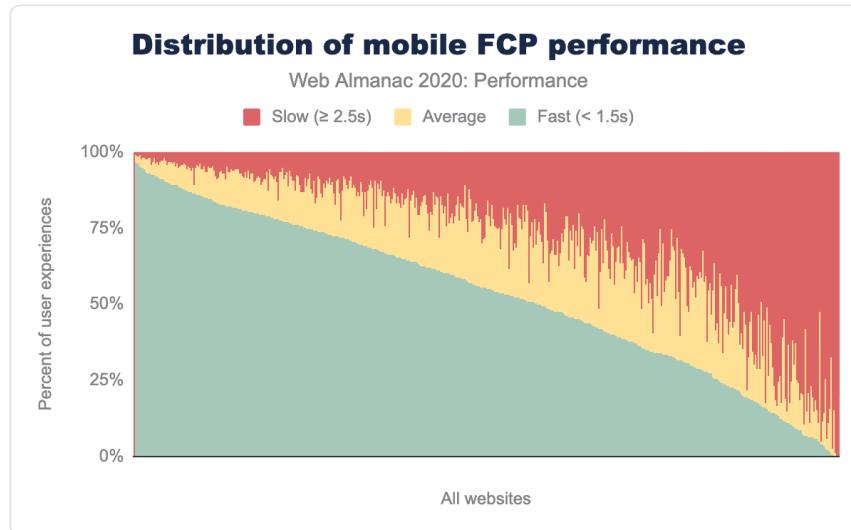


Figure 9.13. Distribution of websites labeled as having fast, average and slow FCP performance on mobile.

In the charts above, the FCP distributions are broken down by desktop and mobile. Comparing to last year, there are noticeably less average FCP readings, while the percentage of fast and slow user experiences has risen no matter the device type. We can still observe the same trend, where mobile users will experience slower FCP more frequently than desktop users. Overall, users are more likely to have a good or poor experience, rather than a mediocre experience.

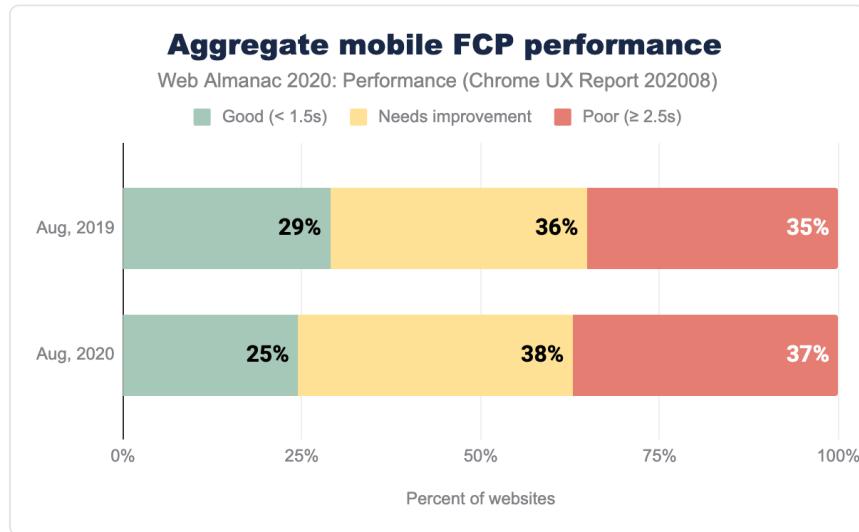


Figure 9.14. A comparison of distribution of websites labeled as having good, needs improvement and poor FCP mobile performance between 2019 and 2020.

Comparing FCP on mobile devices on a year-over-year basis, we observe fewer good experiences and more moderate and poor experiences. 75% of websites have sub-par FCP. We can speculate this high percentage of less than ideal FCP readings is a source of frustration and degraded user experience.

Numerous factors can delay paints, such as server latency (measured by a handful of metrics, such as Time to First Byte (TTFB) and RTT), blocking JavaScript requests, or inappropriate handling of custom fonts to name a few.

FCP by geographic location

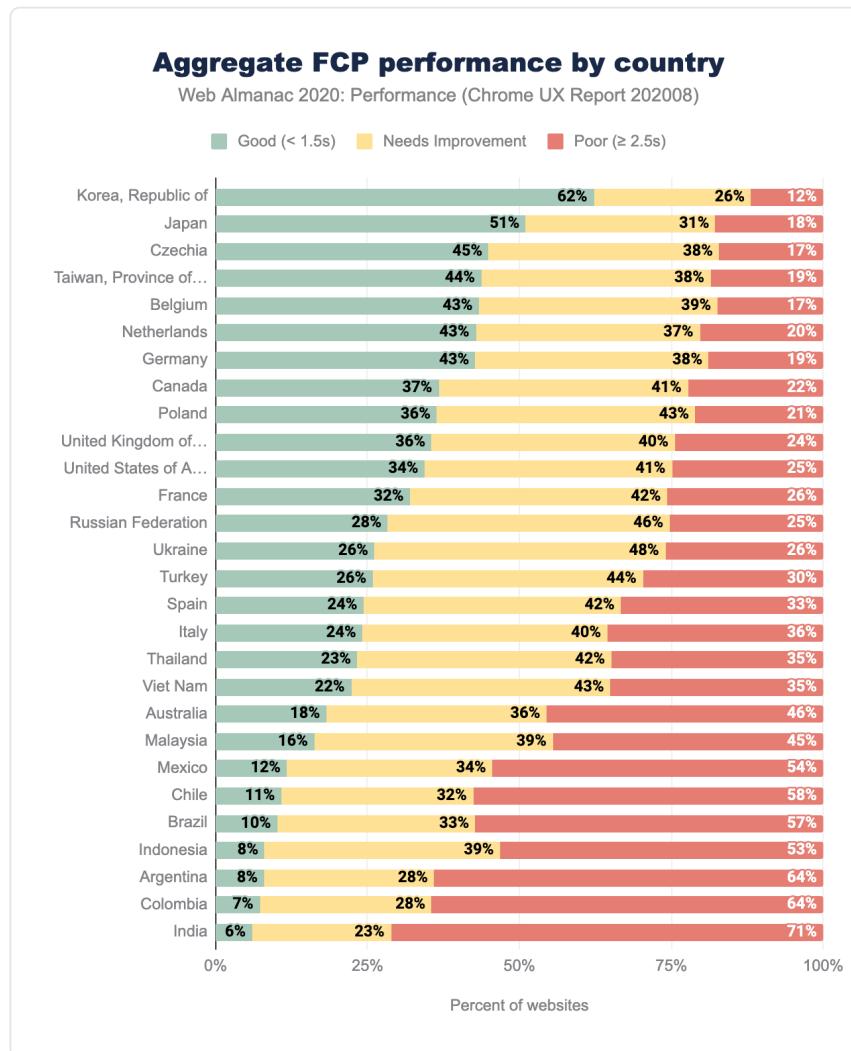


Figure 9.15. Aggregate FCP performance split by country.

Before we dig into the analysis, it is noteworthy to mention that in the 2019 Performance chapter, the thresholds for "good" and "poor" classification were different from 2020. In 2019, sites with FCP below 1s were considered good, while those with FCP above 3s were categorized as poor. In 2020, those ranges shifted to 1.5s for good and 2.5s for poor.

This change means that the distribution would shift towards more "good" and "poor" rated websites. We can observe that trend compared to last year's results, as the percentage of good and poor websites rise. The top ten geographies with the highest rate of fast websites remain relatively unchanged from 2019, with the addition of Czechia and Belgium and the fall of the United States and the United Kingdom. The Republic of Korea leads with 62% of websites reporting fast FCP, nearly doubling since last year (which, again, is likely due to re-categorization of results). Other countries in the top of the ranking also double the number of good experiences.

While the mediocre ("needs improvement") percentage becomes smaller, the number of poorly performing FCP sites rises, which is especially pronounced at the bottom of the ranking with Latin American and South Asian regions.

Again, there are several reasons negatively affecting FCP, such as poor TTFB readings, but it is difficult to prove them without the necessary context. For example, if we were to analyze specific country performance, such as Australia, we surprisingly find it at the lower end. Australia has one of the highest global smartphone penetration levels, one of the fastest mobile networks and relatively high average income levels. We'd easily assume it should be higher up. However, taking into account slow fixed connections, latency and lack of iOS representation in CrUX, its positioning starts making more sense. With an example like this (touched only on the surface), we can see how difficult understanding context for each of the countries would be.

FCP by connection type

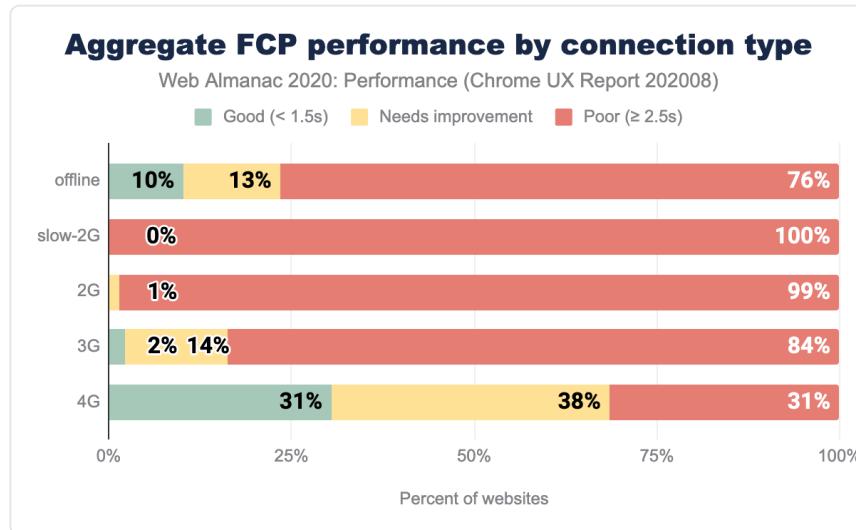


Figure 9.16. Aggregate FCP performance split by connection type.

Similarly to other metrics, FCP is affected by connection speeds. On 3G, only 2% of experiences rate good, while on 4G, 31%. It is not an ideal state of FCP performance, but it has improved since 2019 in some areas, which again might be driven by the change in categorization of good and poor categorization. We see the same rise in the percentage of good websites and poor websites, narrowing the number of moderate ("needs improvement") site experiences.

This trend illustrates the furthering digital divide, where experiences on slower networks and potentially less capable devices are consistently worse. Improving FCP on slow connections directly correlates to enhancing TTFB, which we observe in Aggregate TTFB performance by connection type chart—poor TTFB = poor FCP.

The choice of hosting provider or CDN will have a cascading effect on speed. Making these decisions based on the fastest possible delivery will help in improving FCP and TTFB, especially on slower networks. FCP is also significantly affected by font load time, so ensuring text is visible while web fonts are downloaded is also a worthwhile strategy (especially where on slower connections these resources will be costly to fetch).

Looking at the "offline" statistics, we can deduce that a substantial number of FCP issues are also *not* correlated to the network type. We don't observe significant gains in this category, which we would if that statement was true. It appears would seem rendering is not so much delayed by fetching JavaScript, but it is affected by parsing and execution.

Time to First Byte

Time to First Byte (TTFB) is the time taken from the initial HTML request being made until the first byte arrives back to the browser. Issues with swiftly processing requests can quickly cascade into affecting other performance metrics as they will delay not only paints but also any resource fetching.

TTFB by device

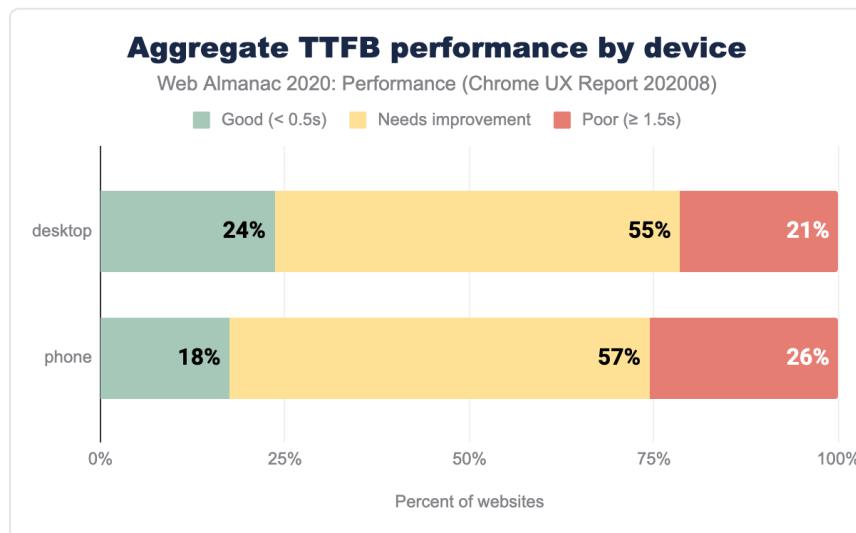


Figure 9.17. Aggregate TTFB performance split by device type.

On desktop, 76% of websites have a "not good" TTFB, while on mobile, that percentage rises to 83%. We might assume that the data portrays how TTFB is often an overlooked metric when it is assumed that most performance measurements and work is concentrated within front-end and visual rendering, not asset delivery and server-side work. High TTFB will have a direct, negative impact on a plethora of other performance signals, which is an area that still needs addressing.

TTFB by geographic location

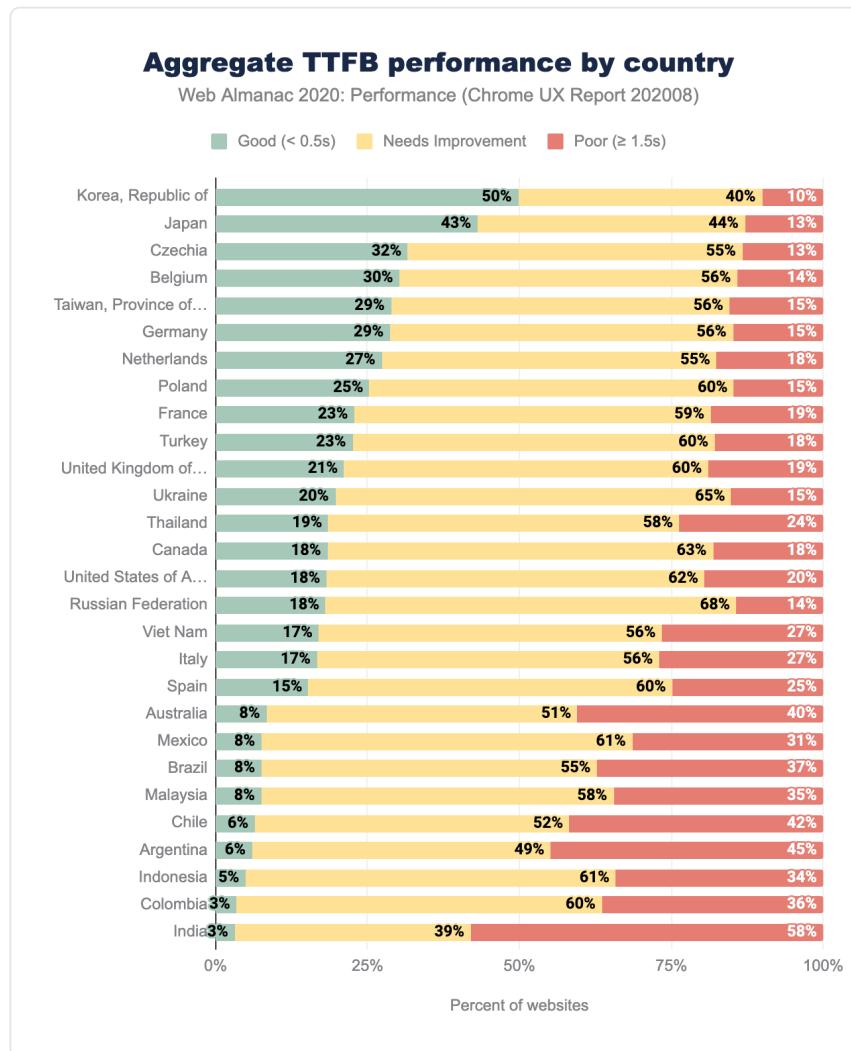


Figure 9.18. Aggregate TTFB performance split by country.

Likening this years' TTFB geo readings to 2019 results again points to more fast websites, but similarly to FCP, the thresholds have changed. Previously, we considered TTFB below 200ms fast, and above 1000ms slow. In 2020, TTFB below 500ms is good and above 1500ms poor. Such generous changes in categorization can explain that we observe significant changes, such as a 36% rise in good website experiences in The Republic of Korea or 22% rise in Taiwan.

Overall, we still observe similar regions, such as Asia-Pacific and selected European locales leading.

TTFB by connection type

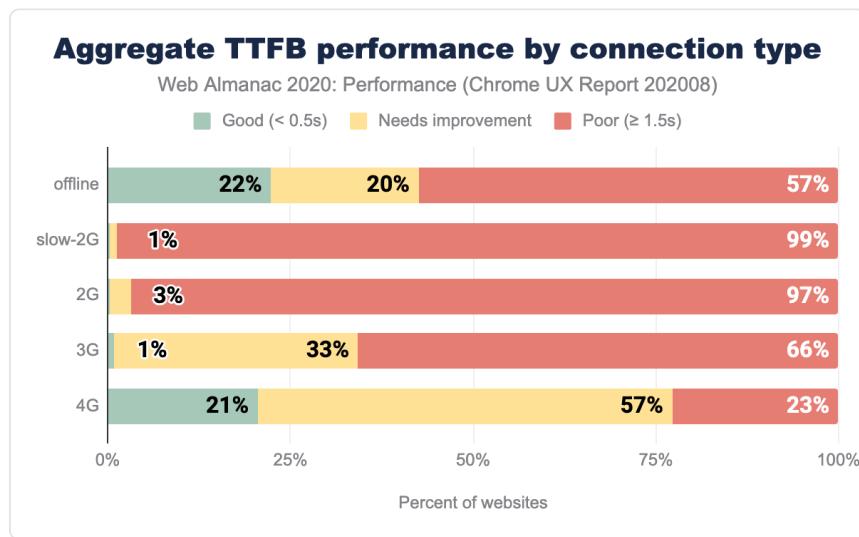


Figure 9.19. Aggregate TTFB performance split by connection type.

TTFB is affected by network latency and connection type. The higher the latency and the slower the connection, the worse TTFB measurements, as we can observe above. Even on mobile connections considered as fast (4G), only 21% of websites have a fast TTFB. There are nearly no sites categorized as quick below 4G speeds.

Looking at the mobile speeds worldwide for December 2018-November 2019, we can see that globally, mobile connections aren't high-speed. Those network speeds and technology standards for cellular networks (such as 5G) are not evenly distributed and affect TTFB. As an example, see this map of networks in Nigeria—most of the country area has 2G and 3G coverage, with little 4G range.

What's surprising is the relatively the same number of good TTFB results between offline and 4G origins. With service workers, we could expect some of the TTFB issues to be mitigated, but that trend is not reflected in the chart above.

Performance Observer usage

There are dozens of different user-centric metrics that can be used to assess websites and applications. However, sometimes the predefined metrics don't quite fit our specific scenarios and needs. The `PerformanceObserver` API allows us to obtain custom metric data obtained with User Timing API, Long Task API, Event Timing API and a handful of other low-level APIs. For example, with their help, we could record the timing transitions between pages or quantify server-side-rendered (SSR) application hydration.

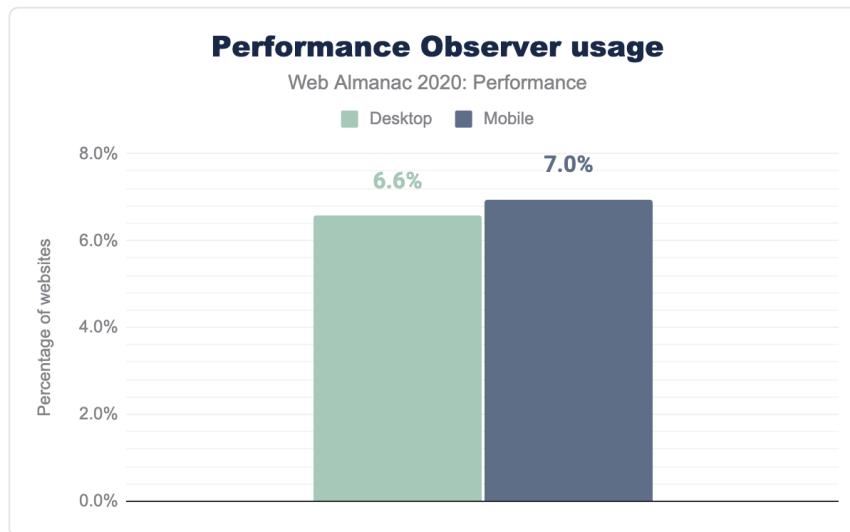


Figure 9.20. Performance Observer usage by device type.

The chart above showcases that Performance Observer is used by 6-7% of tracked sites, depending on device type. Those websites will be leveraging the low-level APIs to create custom metrics, and the `PerformanceObserver` API to collate them, and then potentially use it with other performance reporting tooling. Such adoption rates might indicate the tendency to lean on predefined metrics (for example, coming from Lighthouse), but also are impressive for a relatively niche API.

Conclusion

User experience is not only a spectrum but also depends on a wide variety of factors. To attempt understanding the state of performance without excluding the sub-par, underprivileged experiences, we must approach it intersectionally. Each website visit tells a

story. Our personal and country-level socioeconomic status dictates the type of device and internet provider we can afford. The geopositioning of where we live affects latency (we Australians feel this pain regularly), and the economy dictates available cellular network coverage. What websites do we visit? What do we visit them for? Context is critical to not only analyzing data but also developing necessary empathy and care in building accessible, fast experiences for all.

On the surface, we have seen optimistic signals about the new Core Web Vitals performance metrics. At least half of the experiences are good across both desktop and mobile devices, if we don't narrow down to consistently poor experiences on slower networks for Largest Contentful Paint. While the newer metrics might suggest that there's an ongoing uptake in addressing performance issues, the lack of significant improvements in First Contentful Paint and Time to First Byte is sobering. Here the same network types are most disadvantaged as with Largest Contentful Paint, as well as fast connections and desktop devices. The Performance Score also portrays a decline in speed (or perhaps, a more accurate portrayal than what we measured in the past).

What the data shows us, is that we must keep investing in improving performance for scenarios (such as slower connectivity) that we often don't experience due to multiple aspects of our privilege (middle to high-income countries, high pay and new, capable devices). It also highlights that there's still plenty of work to be done in the areas of speeding up initial paints (LCP and FCP) and asset delivery (TTFB). Often, performance feels like an inherently front-end issue, while numerous significant improvements can be achieved on the back-end and through appropriate infrastructure choices. Again, user experience is a spectrum that depends on a variety of factors, and we need to treat it holistically.

New metrics bring new lenses to analyze user experience through, but we must not forget existing signals. Let's focus on moving the needle in the areas that need the most improvement and will result in positive shifts in experience for most underserved. Fast and accessible internet is a human right.

Author



Karolina Szczur

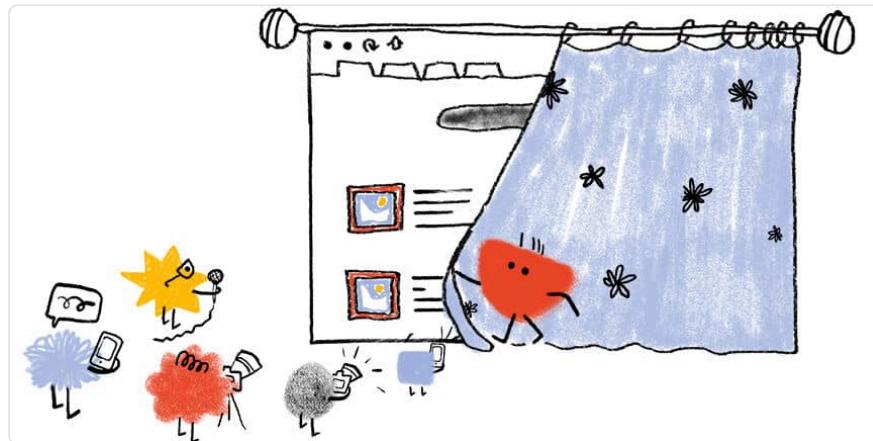
Twitter @fox GitHub thefoxis

Karolina is a Product Design Lead at Calibre⁴¹, working on creating the most comprehensive speed monitoring platform. She curates Performance Newsletter⁴², your source of performance news and resources. Karolina also frequently writes⁴³ about how performance affects user experience.

41. <https://calibreapp.com/>
42. <https://perfemail/>
43. <https://calibreapp.com/blog/category/web-platform>

Part II Chapter 10

Privacy



Written by Yana Dimova

Reviewed by Laurent Devernay

Analyzed by Yana Dimova and Max Ostapenko

Introduction

This chapter of the Web Almanac gives an overview of the current state of privacy on the web. This topic has been increasing in popularity recently and has raised awareness on the users' side. The need for guidelines has been met with various regulations (such as GDPR in Europe, LGPD in Brazil, CCPA in California to name but a few). These aim to increase the accountability of data processors and their transparency towards users. In this chapter, we discuss the prevalence of online tracking with different techniques and the adoption rate of cookie consent banners and privacy policies by websites.

Online tracking

Third-party trackers collect user data to build up profiles of the user's behavior to be monetized for advertising purposes. This raises privacy concerns with users on the web, which resulted in the emergence of various tracking protections. However, as we will see in this section, online

tracking is still widely used. Not only does it have a negative impact on privacy, online tracking has a huge impact on the environment and avoiding it can lead to better performance.

We examine the prominence of the most common types of third-party tracking, namely by means of third-party cookies and the use of fingerprinting. Online tracking is not limited to just these two techniques, new ones keep arising to circumvent existing countermeasures.

Third-party trackers

We use WhoTracksMe's tracker list to determine the percentage of websites that issue a request to a tracker. As shown in the following figure, we have found that at least one tracker is present on roughly 93% of websites.

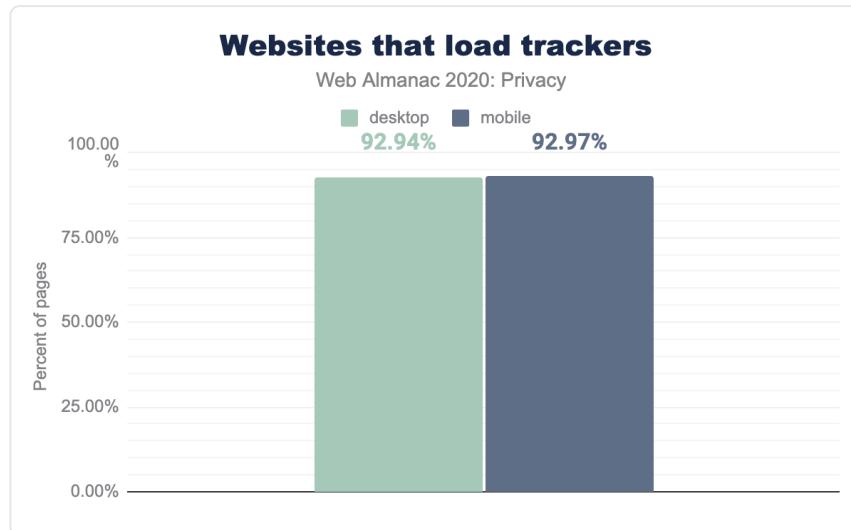


Figure 10.1. Websites including at least one tracker

We examined the most widely used trackers and plot the prevalence of the 10 most popular ones.

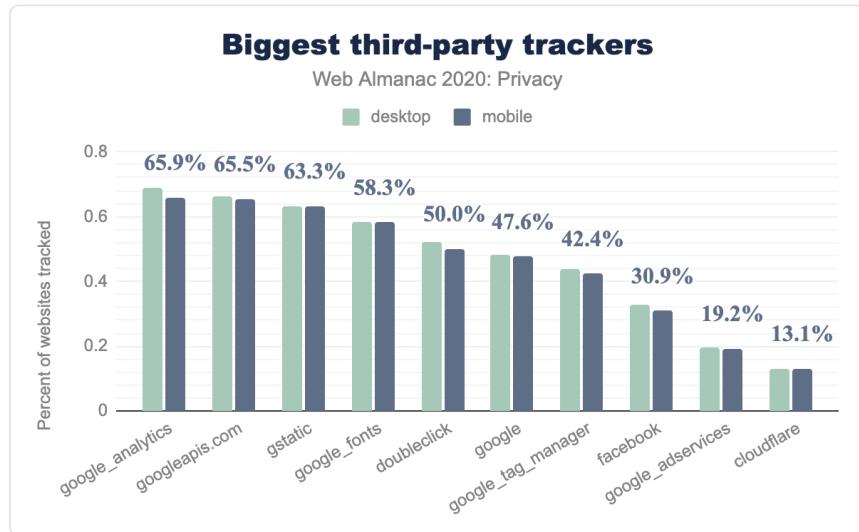


Figure 10.2. Top 10 Trackers

The largest player on the online tracking market is without doubt Google, with eight of its tracking domains present in the top 10 trackers and prevalent on at least 70% of websites. They are followed are Facebook and Cloudflare—though the latter is probably more reflective of the popularity of them as a hosting site.

WhoTracksMe's tracker list also defines categories that the trackers belong to. The following figure shows the distribution of the different categories for the 100 largest trackers.

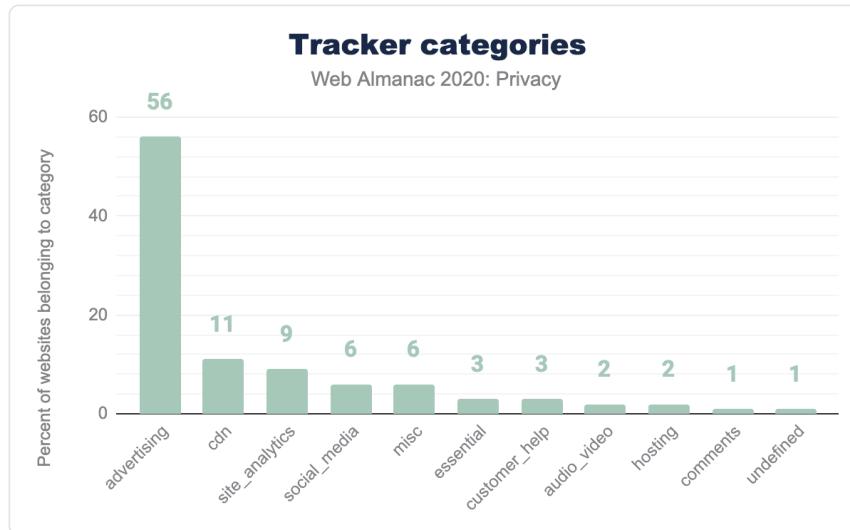


Figure 10.3. Categories of the 100 most popular trackers

Nearly 60% of the most popular trackers are advertising-related. This could be due to the profitability of the online advertising market.

Cookies

We looked into the most popular cookies being set on websites in HTTP's response header, according to their name and domain.

Domain	Cookie Name	Websites
doubleclick.net	<i>test_cookie</i>	24%
facebook.com	<i>fr</i>	10%
youtube.com	<i>VISITOR_INFO1_LIVE</i>	10%
youtube.com	<i>YSC</i>	10%
doubleclick.net	<i>IDE</i>	9%
doubleclick.net	<i>unknown</i>	9%
youtube.com	<i>GPS</i>	9%
doubleclick.net	<i>unknown</i>	8%
google.com	<i>NID</i>	6%
doubleclick.net	<i>unknown</i>	6%

Figure 10.4. Top cookies on desktop sites

Domain	Cookie Name	Websites
doubleclick.net	<i>test_cookie</i>	32%
doubleclick.net	<i>IDE</i>	21%
facebook.com	<i>fr</i>	10%
youtube.com	<i>VISITOR_INFO1_LIVE</i>	10%
youtube.com	<i>YSC</i>	10%
google.com	<i>NID</i>	10%
youtube.com	<i>GPS</i>	8%
doubleclick.net	<i>DSID</i>	7%
yandex.ru	<i>yandexuid</i>	6%
yandex.ru	<i>i</i>	6%

Figure 10.5. Top cookies on mobile sites

As you can see, Google's tracking domain "doubleclick.net" sets cookies on roughly a quarter of websites on a mobile client and a third of all websites on a desktop client. Again, nine out of the ten most popular cookies on desktop client and seven out of ten on mobile are set by a Google domain. This is a lower bound for the number of websites the cookie is set on, since we are only counting cookies set via an HTTP header—a large number of tracking cookies are set by using third-party scripts.

Fingerprinting

Another widely-used tracking technique is fingerprinting. This consists of collecting different kinds of information about the user with the goal of building a unique "fingerprint" for them. Different types of fingerprinting are used on the web by trackers. Browser fingerprinting use characteristics specific to the browser of the user, relying on the fact that the chance of another user having the exact same browser set-up is fairly small if there are a large enough number of variables to track. In our crawl, we examined the presence of the FingerprintJS library, which provides browser fingerprinting as a service.

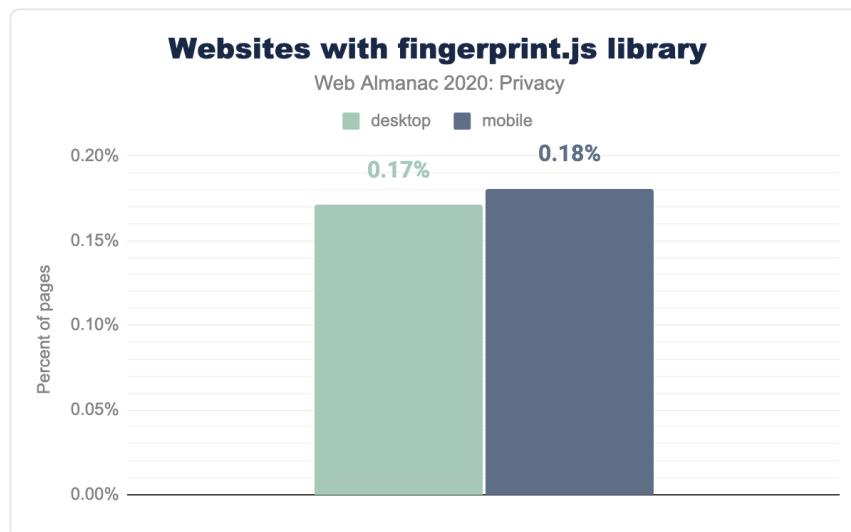


Figure 10.6. Websites using FingerprintJS

Although the library is present on only a small percentage of websites, the persistent nature of fingerprinting means even small usage can have a big impact. Furthermore, FingerprintJS is not the only attempt at fingerprinting. Other libraries, tools and native code can also serve this purpose, so this is just one example.

Consent Management Platforms

Cookie consent banners have become common now. They increase transparency towards cookies and often allowing users to specify their cookie choices. While a lot of websites opt for using their own implementation of cookie banners, third-party solutions called *Consent Management Platforms* have recently emerged. The platforms provide an easy way for websites to collect user's consent for different types of cookies. We see that 4.4% of websites use a consent management platform to manage cookie choices on desktop clients, and 4.0% on mobile clients.

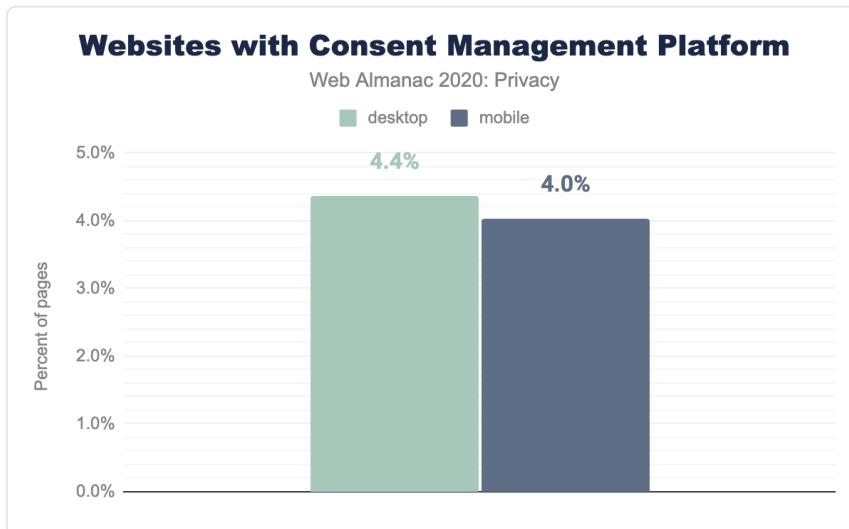


Figure 10.7. Websites using a consent management platform

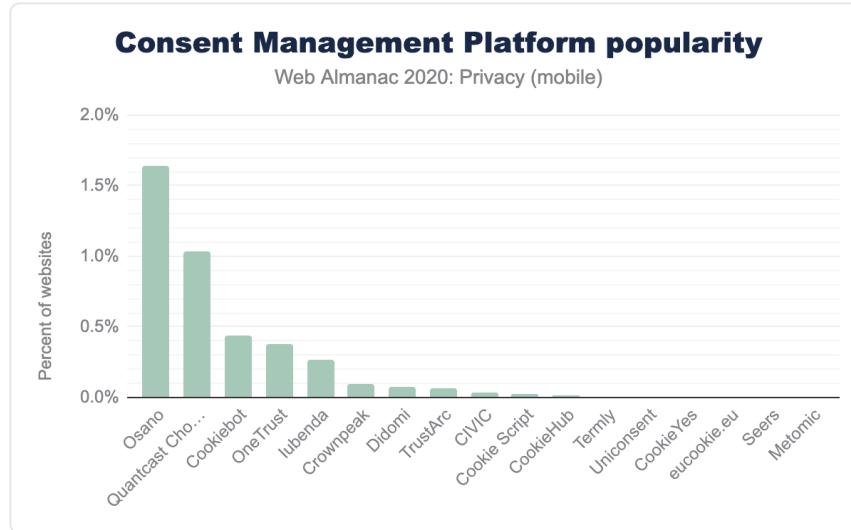


Figure 10.8. Popularity of consent management platform

When looking at the popularity of the different consent management solutions, we can see that Osano and Quantcast Choice are the leading platforms.

IAB Europe's Transparency Consent Framework

IAB Europe, the Interactive Advertising Bureau, is a European association for digital marketing and advertising. They proposed a Transparency Consent Framework (TCF) as a GDPR-compliant solution to obtain users' consent about their digital advertising preferences. The implementation provides an industry standard for communication between publishers and advertisers about consumer consent.

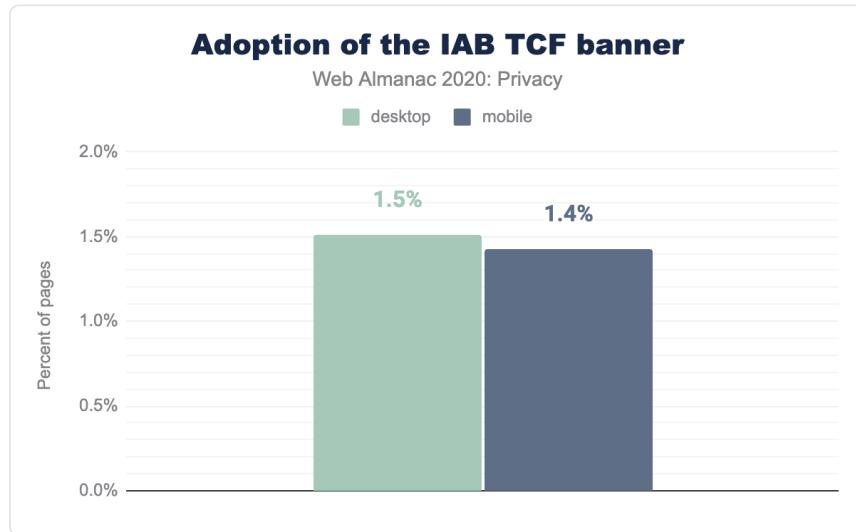


Figure 10.9. Adoption rate of TCF banner

While our results show that the TCF banner is not yet the "industry standard", it is a step in the right direction. Considering the main target group of IAB Europe is in fact European publishers, and our crawl is global, having an adoption rate on 1.5% of websites on desktop client and 1.4% on mobile is not too bad.

Privacy Policies

Privacy policies are widely used by websites to meet legal obligations and increase transparency towards users about data collection practices. In our crawl, we searched for keywords indicating the presence of a privacy policy text on each visited website.

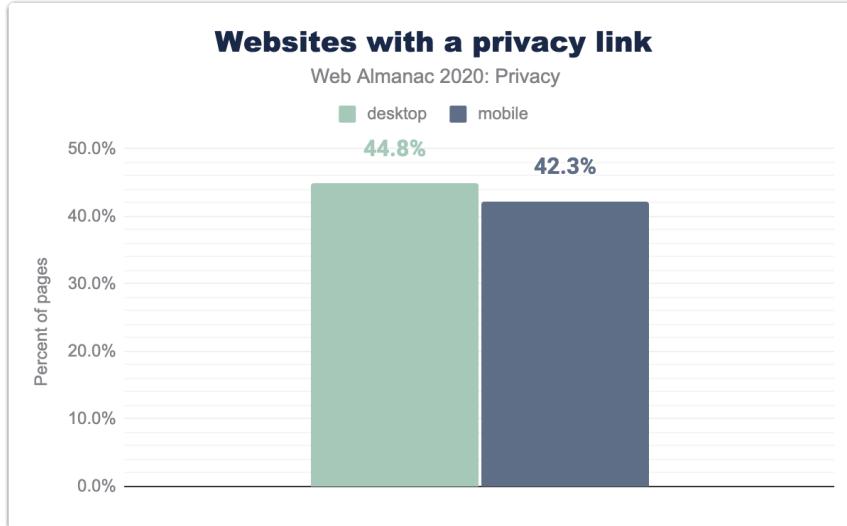


Figure 10.10. Websites that have a privacy policy

The results show that almost half of the websites in the dataset have included a privacy policy, which is positive. However, studies have shown that the majority of internet users do not bother reading privacy policies and when they do, they lack understanding due to the length and complexity of most privacy policy texts. Still having a policy at all is a step in the right direction!

Conclusion

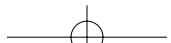
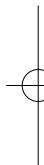
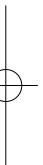
This chapter has shown that third-party tracking remains prominent on both desktop and mobile clients, with Google tracking the largest percentage of websites. Consent Management Platforms are used on a small percentage of websites; however a lot of websites implement their own cookie consent banners.

Lastly, roughly half of the websites include a privacy policy, which benefits greatly transparency towards users about data processing practices. This is undoubtedly a step forward but there is a lot still to be done. Outside of this analysis we know that privacy policies are hard to read and understand and cookie consent banners manipulate users into consent.

For the web to truly respect users, privacy has to be a part of conception, not an afterthought. Regulation is a good thing in this regards, and it is reassuring to see an increase in privacy regulation worldwide. Privacy by design should be the norm, rather than deploying policies and tools in order to meet minimum legal requirements and avoid financial penalties.

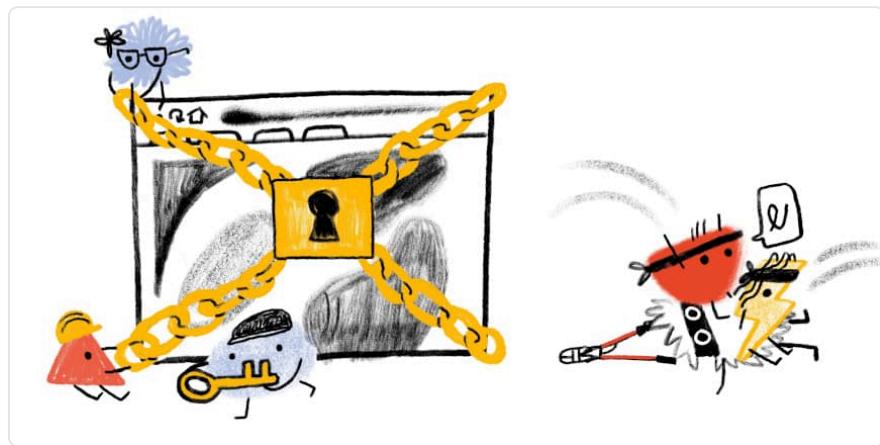
Author**Yana Dimova** [ydimova](#)

Yana Dimova is a PhD student at KU Leuven university in Belgium, working on privacy and web security.



Part II Chapter 11

Security [UNEDITED]



Written by Tom Van Goethem, Nurullah Demir, and Barry Pollard

Reviewed by Caleb Queern and Edmond W. W. Chan

Analyzed by Tom Van Goethem and Nurullah Demir

Introduction

In many ways, 2020 has been an extraordinary year. As a result of the global pandemic, our day-to-day lives have changed drastically: instead of meeting in person with friends and family, many have to rely on social media to keep in touch. This has led to a significant increase in traffic volumes for many different applications, as a result of the increased amount of time that users spend online. This also means that security has never been more important to ensure that the information we share online on various platforms remains secure.

Many of the platforms and services that we use on a daily basis strongly rely on web resources, ranging from cloud-based APIs, microservices, and most importantly, web applications. Keeping these systems secure is a non-trivial task. Fortunately, throughout the past decade, web security research has been continuously advancing. On the one hand researchers are discovering new types of attacks and sharing the results with the wider community to raise awareness. On the other hand, many engineers and developers have been tirelessly working to

provide website operators with the right set of tools and mechanisms that can be used to prevent or minimize the consequences of attacks.

In this chapter, we explore the current state-of-practice for security on the Web. By analyzing the adoption of various security features in depth and at a large scale we gather insights on the different ways that website owners apply these security mechanisms, driven by the incentive to protect their users.

We not only look at the adoption of security mechanisms in individual websites. We analyze how different factors, such as the technology stack that is used to build a website, affect the prevalence of security headers, and thus improve overall security. Furthermore, it is safe to say that ensuring a website is secure requires a holistic approach covering many different facets. Therefore we also evaluate other aspects, such as the patching practices for various widely used web technologies.

Methodology

Throughout this chapter, we report on the adoption of various security mechanisms on the web. This analysis is based on data that was collected for the homepage of 5.6M desktop domains and 6.3M mobile domains. Unless explicitly stated otherwise, the reported numbers are based on the mobile dataset, as it is larger in size. Because most websites are included in both datasets, the resulting measurements are mostly similar. Any significant difference between the two datasets are reported throughout the text or is apparent from the figures. For more information on how the data has been collected, please refer to the methodology.

Transport security

The last year has seen a continuation of the growth of HTTPS on websites. Securing the transport layer is one of the most core parts of web security—unless you can be confident the resources downloaded for a website have not been altered in transit, and that you are transporting data to and from the website you think you are, any certainties about the website security are effectively rendered null and void.

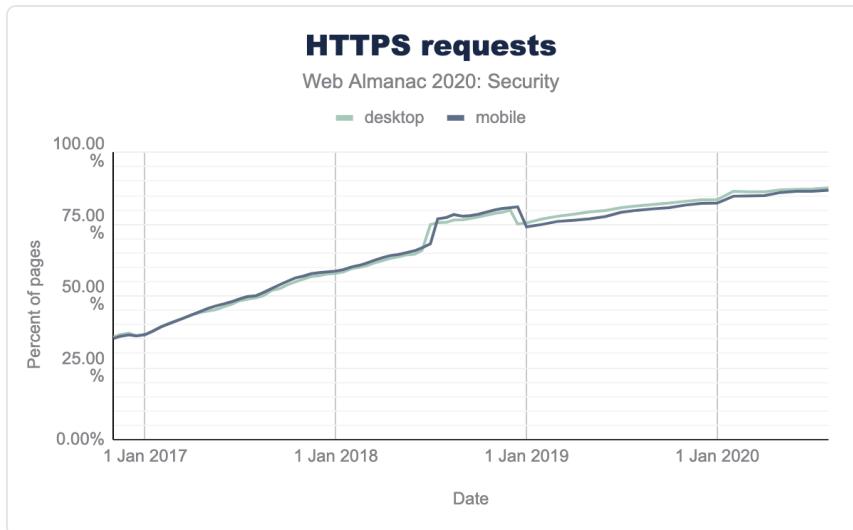
Moving our web traffic to HTTPS, and eventually marking HTTP as non-secure is being driven by web browsers only allowing powerful new features to the secure context (the carrot) while also increasing warnings shown to users when unencrypted connections are used (the stick).

The effort is paying off and we are now seeing 87.70% of requests on desktop and 86.90% of requests on mobile being served over HTTPS.

86.90%

Figure 11.1. The percentage of requests that use HTTPS on mobile.

One slight concern as we reach the end of this goal, is a noticeable "leveling off" of the impressive growth of the last few years. Unfortunately the long tail of the internet means older legacy sites are not maintained and may never be run over HTTPS, meaning they will eventually become inaccessible to most users.



*Figure 11.2. Percentage of requests using HTTPS.
(Source: HTTP Archive)*

Whilst the high volume of requests is encouraging, these can often be dominated by third-party requests and services like Google Analytics, fonts or advertisements. Websites themselves can lag, but again we see encouraging use with between 73% and 77% of sites now being served over HTTPS.

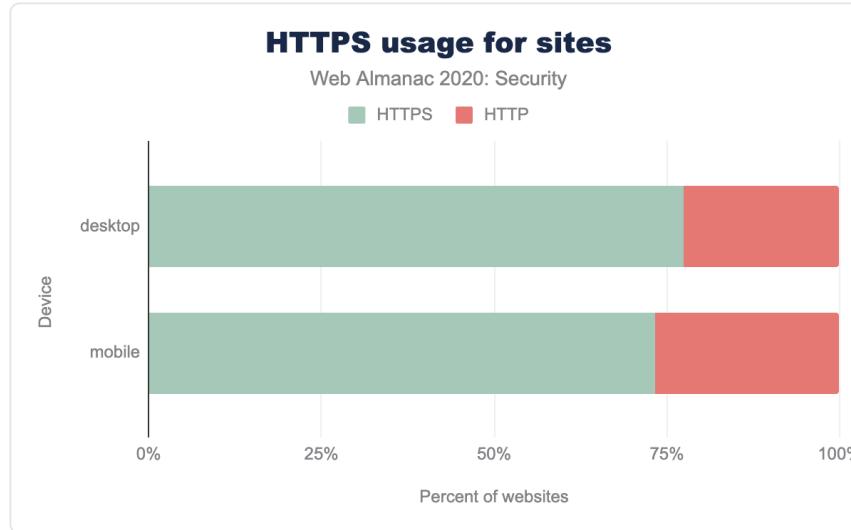


Figure 11.3. HTTPS usage for sites

Protocol versions

As HTTPS is now well and truly the norm, the challenge moves from having any sort of HTTPS, to ensuring that secure versions of the underlying TLS (Transport Layer Security) protocol are being used. TLS needs maintenance as versions become older, vulnerabilities are found and compute increase making attacks more achievable.

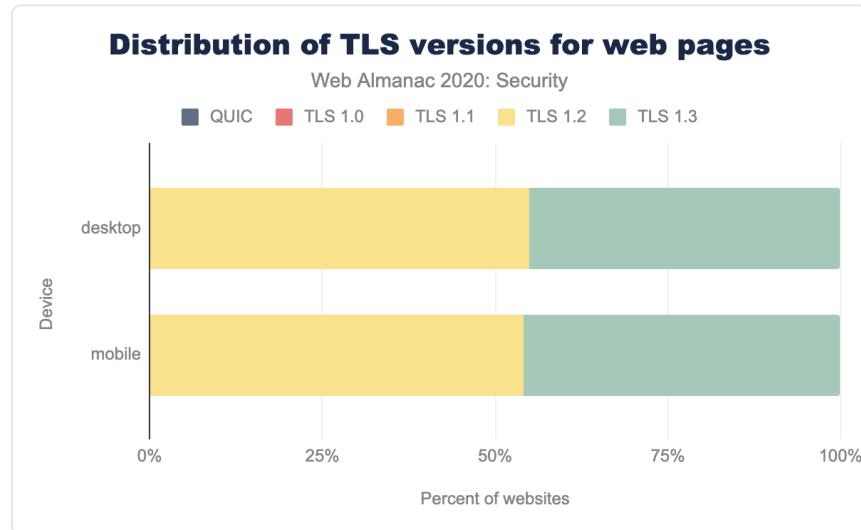


Figure 11.4. TLS versions usage for sites

It's still surprising to us, that TLSv1.0 usage is basically zero and encouraging that the public web has embraced more secure protocols so definitely. These figures are a slight improvement on last year's protocol analysis with an approximately 5% increase in TLSv1.3 usage, and the corresponding drop in TLSv1.2. That seems a small increase and it would seem like the high usage of the relatively new TLSv1.3 noted last year was likely due to the initial support from large CDNs. Therefore making more significant progress in TLSv1.3 adoption will likely take a long time as those still using TLSv1.2 are potentially managing this themselves or with a hosting provider that does not yet support this.

Cipher suites

Within TLS there are a number of cipher suites that can be used with varying levels of security. The best ciphers support forward secrecy key exchange, meaning even if the servers keys are compromised old traffic that used those keys cannot be decrypted.

In the past, newer versions of TLS added support for newer ciphers but rarely removed older versions. This is one of the reasons TLSv1.3 is more secure as it does a large clear down of older ciphers currently with the popular OpenSSL library only supporting five secure ciphers in this version—all of which support forward secrecy. This prevents downgrade attacks where a less secure cipher is forced to be used.

98.03%

Figure 11.5. Mobile sites using forward secrecy.

All sites should be using forward secrecy ciphers and it is good to see 98.14% of desktop sites and 98.03% of mobile sites using ciphers with forward secrecy.

Assuming forward secrecy is a given, the main choice in selecting a cipher is between the level of encryption - higher key sizes will take longer to break, but at the cost of more compute intensive to encrypt and decrypt the connection, particularly for initial connection. For the block cipher mode GCM should be used and CBC is considered weak due to padding attacks.

For the widely used Advanced Encryption Standard (AES) key sizes of 128-bit and 256-bit encryption are common. 128-bit is still sufficient for most sites, though 256-bit would be preferred.

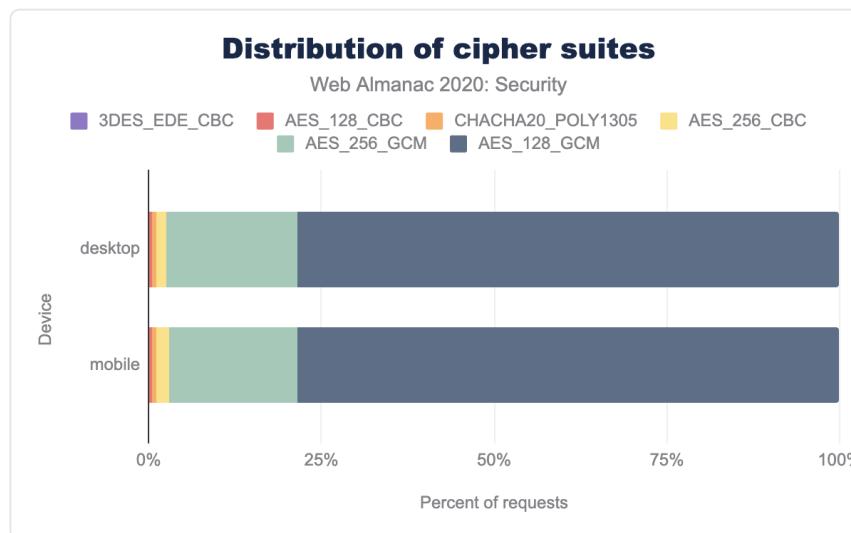


Figure 11.6. Distribution of cipher suites

We can see from the above chart that AES_128_GCM is the most common and is used by 78.4% of desktop and mobile sites. AES_256_GCM is used by 19.1% of desktop and 18.5% of mobile sites with the other sites likely being the ones on older protocols and cipher suites.

One important point to note is that our data is based on running Chrome to connect to a site, and it will use a single protocol cipher to connect. Our methodology does not allow us to see the full range of protocols and cipher suites supported, and only the one that was actually used for that connection. For that information we would need to look at other sources like SSL Pulse from SSL Labs, but with most modern browsers now supporting similar TLS capabilities the above data is what we would expect the vast majority of users to use.

Certificate Authorities

Next we will look at the Certificate Authorities (CAs) issuing the TLS certificates used by the sites we have crawled. Last year's chapter, looked at the requests for this data, but that will be dominated by popular third-parties like Google (who also dominate again this year from that metric), so this year we are going to present the CAs used by the websites themselves, rather than all the other request they load.

Issuer	Desktop	Mobile
Let's Encrypt Authority X3	44.65%	46.42%
Cloudflare Inc ECC CA-3	8.49%	8.69%
Sectigo RSA Domain Validation Secure Server CA	8.27%	7.91%
cPanel, Inc. Certification Authority	4.71%	5.06%
Go Daddy Secure Certificate Authority - G2	4.30%	3.66%
Amazon	3.12%	2.85%
DigiCert SHA2 Secure Server CA	2.04%	1.78%
RapidSSL RSA CA 2018	2.01%	1.96%
CloudFlare Inc ECC CA-2	1.95%	1.70%
AlphaSSL CA - SHA256 - G2	1.35%	1.30%

Figure 11.7. Top 10 certificate issuers for websites.

It is no surprise to see Let's Encrypt well in the lead easily taking the top spot; its combination of free and automated certificates is proving a winner with both individual website owners and platforms. Cloudflare similarly offers free certificates for its customers taking the number two and number nine position. What is more interesting there is that it is the ECC Cloudflare issuer that is being used. ECC certificates are smaller and so more efficient than RSA certificates but can be complicated to deploy as support is not universal and managing both certificates often

requires extra effort. This is the benefit of a CDN or hosted provider if they can manage this for you like Cloudflare does here. Browsers that support ECC (like the Chrome browser we use in our crawl) will use that, and older browsers will use RSA.

Browser enforcement

Although having a secure TLS configuration is paramount to defend against cryptographic attacks, additional protections are still needed to protect web users from adversaries on the network. For instance, as soon as the user loads any website over HTTP, an attacker can inject malicious content to, for instance, make requests to other sites.

Even when sites are using the strongest ciphers and latest protocols, an adversary can still use SSL stripping attacks to trick the victim's browser into believing that the connection is over HTTP instead of HTTPS. Moreover, without adequate protections in place, a user's cookies can be attached in the initial plaintext HTTP request, allowing the attacker to capture them on the network.

To overcome these issues, browsers have provided an additional feature that can be enabled.

HTTP Strict Transport Security

The first one is HTTP Strict Transport Security (HSTS), which can easily be enabled by setting a response header consisting of several attributes. For this header we find an adoption rate of 16.88% within the mobile homepages. Of the sites that enable HSTS, 92.82% do so successfully. That is, the max-age attribute (which determines how many seconds the browser should *only* visit the website over HTTPS) has a value larger than 0.

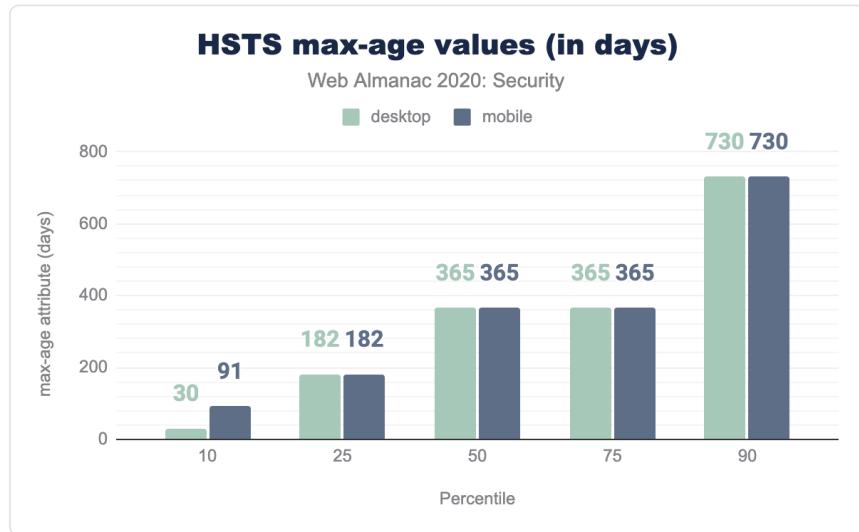


Figure 11.8. HSTS `max-age` values (in days).

Looking at the different values for this attribute, we can clearly see that the majority of websites are confident that they will be running over HTTPS in the considerable future: more than half request the browser to use HTTPS for at least 1 year.

1,000,000,000,000,000 years

Figure 11.9. The largest known HSTS max-age.

One website might have been a bit too enthusiastic about how long their site will be available over HTTPS and set a `max-age` attribute value that translates to 1,000,000,000,000,000 years. Ironically, browsers do not handle such a large value well, and actually disable HSTS for that site.

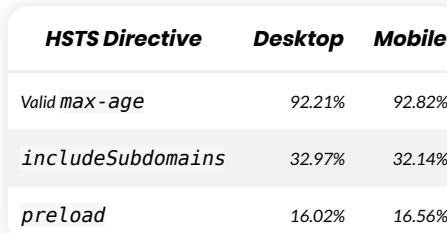


Figure 11.10. Usage of HSTS directives.

It is encouraging to see that the adoption of the other attributes is growing compared to last year: `includeSubdomains` is now at 32.14% and `preload` at 16.56% of HSTS policies.

Cookies

From a security point of view, the automatic inclusion of cookies in cross-site requests can be seen as the main culprit of several classes of vulnerabilities. If a website does not have the adequate protections in place (e.g. requiring a unique token on state-changing requests), they may be susceptible to Cross-Site Request Forgery (CSRF) attacks. As an example, an attacker may issue a POST request in the background, without the user being aware of it, for instance, change the password of an unwitting visitor. If the user is logged in, the browser would normally automatically include the cookies in such a request.

Several other types of attacks rely on the inclusion of cookies in third-party, such as Cross-Site Script Inclusion (XSSI) and various techniques in the XS-Leaks vulnerability class. Furthermore, because the authentication of users is often only done through cookies, an attacker could impersonate a user by obtaining their cookies. This could be done in a man-in-the-middle (MitM) attack, tricking the user to make an authenticated over an insecure channel. Alternatively, by exploiting a cross-site scripting (XSS) vulnerability, the attacker could leak the cookies by accessing `document.cookie` through the DOM.

To defend against the threats posed by cookies, website developers can make use of three attributes that can be set on cookies: `HttpOnly`, `Secure` and `SameSite`. The first prevents the cookie from being accessed from JavaScript, preventing an adversary from stealing them in an XSS attack. Cookies that have the `Secure` attribute set will only be sent over a secure HTTPS connection, preventing them to be stolen in a MitM attack.

The attribute that was introduced most recently, `SameSite`, can be used to restrict how cookies are sent in a cross-site context. The attribute has three possible values: `None`, `Lax`, and `Strict`. Cookies with `SameSite=None` will be sent in all cross-site requests, whereas cookies with the attribute set to `Lax` will only be sent in navigational requests, e.g. when the

user clicks a link and navigates to a new page. Finally, cookies with the `SameSite=Strict` attribute will only be sent in a first-party context.

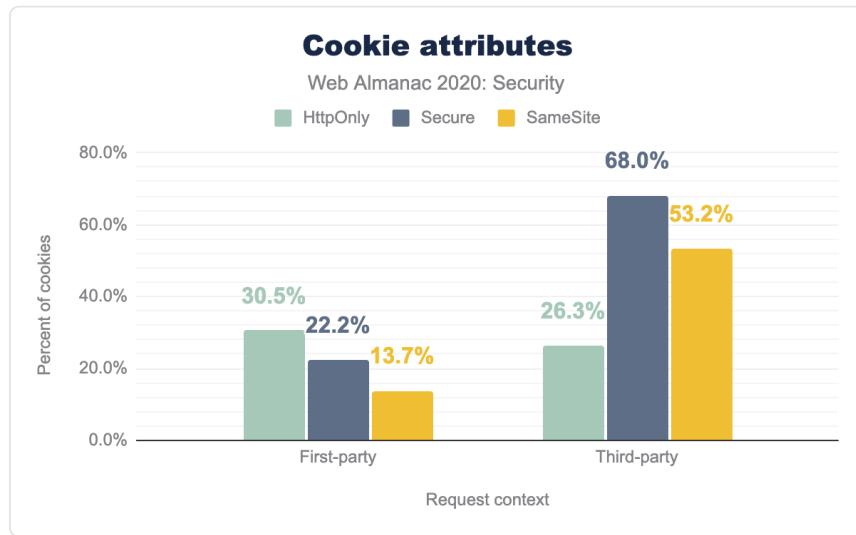


Figure 11.11. Cookie attributes.

Our results, which are based on 25M first-site cookies and 115M third-party cookies, shows that the usage of the cookie attributes strongly depends on the context in which they are set. We can observe a similar usage of the `HttpOnly` attribute on cookies for both first-party (30.5%) and third-party (26.3%) cookies.

However, for the `Secure` and `SameSite` attributes we see a significant difference: The `Secure` attribute is present on 22.2% of all cookies set in a first-party context, whereas 68.0% of all cookies set by third-party requests on mobile homepages have this cookie attribute. Interestingly, for desktop pages, only 35.2% of the third-party cookies had the attribute.

For the `SameSite` attribute, we can see a significant increase in their usage, compared to last year, when only 0.1% of the cookies had this attribute. As of August 2020, we observed that 13.7% of the first-party cookies and 53.2% of third-party cookies have the `SameSite` attribute set.

Presumably, this significant change in adoption is related to the decision of Chrome to make `SameSite=Lax` the default option. This is confirmed by looking more closely at the values set in the `SameSite` attribute: the majority of third-party cookies (76.5%) have the attribute value set to `None`. For first-party cookies, the share is lower, at 48.0%, but still significant. It's important to note that because the crawler does not log in to websites, the cookies used to authenticate users may be different.

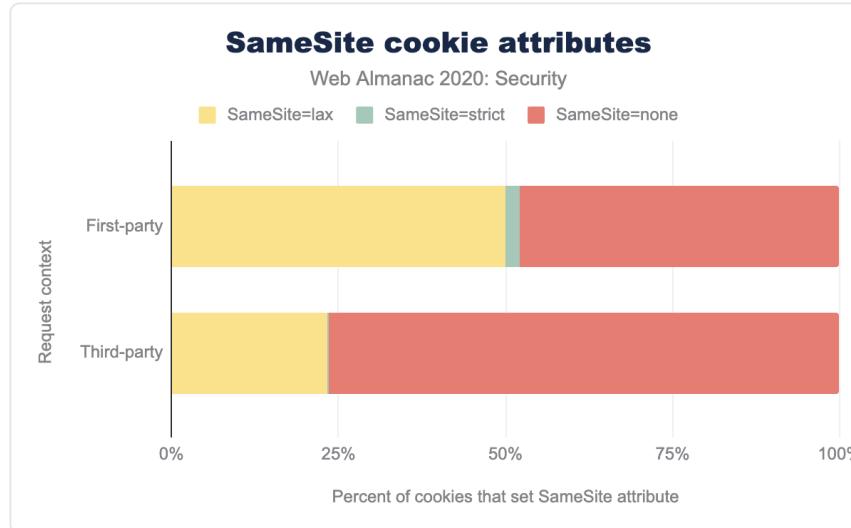


Figure 11.12. Same site cookie attributes.

One additional mechanism that can be used to protect cookies is to prefix the name of the cookie with `Secure-` or `Host-`. Cookies with any of these two prefixes will only be stored in the browser if they have the `Secure` attribute set. The latter imposes an additional restriction, requiring the `Path` attribute to be set to `/` and preventing the use of the `Domain` attribute. This prevents attackers from overriding the cookie with other values, in an attempt to perform a session fixation attack.

The usage of these prefixes is relatively small: in total we found 4,433 (0.02%) first-party cookies that were set with the `Secure-` prefix and 1,502 (0.01%) with the `Host-` prefix. For cookies set in a third-party context, the relative number of prefixed cookies is similar.

Content inclusion

Modern web applications include a large variety of third-party components, ranging from JavaScript libraries, to video players, to external plugins. From a security perspective, including potentially untrusted content in your web page may pose various threats, such as cross-site scripting in case a malicious JavaScript file gets included. To defend against these threats, browsers have several mechanisms that can be used to limit from which sources content can be included, or to impose limitations on the included content.

Content Security Policy

One of the predominant mechanisms to indicate to the browser which origins are allowed to load content, is the `Content-Security-Policy` (CSP) response header. Through numerous directives, a website administrator can have fine-grained control over how content can be included. For instance, the `script-src` directive indicates from which origins scripts can be loaded. Overall, we found that a CSP header was present on 7.23% of all pages, a notable increase of 53% from last year, when CSP adoption was at 4.73% for mobile pages.

Directive	Desktop	Mobile
<code>upgrade-insecure-requests</code>	61.61%	61.00%
<code>frame-ancestors</code>	55.64%	56.92%
<code>block-all-mixed-content</code>	34.19%	35.61%
<code>default-src</code>	18.51%	16.12%
<code>script-src</code>	16.99%	16.63%
<code>style-src</code>	14.17%	11.94%
<code>img-src</code>	11.85%	10.33%
<code>font-src</code>	9.75%	8.40%

Figure 11.13. Most common directives used in CSP policies.

Interestingly, when we look at the most commonly used directives in CSP policies, the most common directive is `upgrade-insecure-requests`, which is used to signal to the browser that any content that is included from an insecure scheme should instead be accessed via a secure HTTPS connection to the same host.

For instance, the image that would be fetched over an insecure connection in `` will instead be fetched over HTTPS when the `upgrade-insecure-requests` directive is present.

This is particularly helpful as browsers block mixed content: for pages that are loaded over HTTPS, content that is included from HTTP would be blocked without the `upgrade-insecure-requests` directive. The adoption of this directive is likely much higher relative to the others as it is a good starting point for a content security policy as it is unlikely to break content and is easy to implement.

The CSP directives that indicate from which sources content can be included (the `*-src` directives), have a much lower adoption: only 18.51% of the CSP policies served on desktop pages and 16.12% on mobile pages. One of the reasons for this, is that web developers are facing many challenges in the adoption of CSP. Although a strict CSP policy can provide significant security benefits well beyond thwarting XSS attacks, an ill-defined one may prevent certain content from loading.

To allow web developers to evaluate the correctness of their CSP policy, there also exists a non-enforcing alternative, which can be enabled by defining the policy in the `Content-Security-Policy-Report-Only` response header. The prevalence of this header is fairly small: 0.85% of desktop and mobile pages. It should be noted however that this is often a transitional header and so the percentage likely indicates the sites that intend to transition to using CSP and are only using the Report-Only header for a limited amount of time.

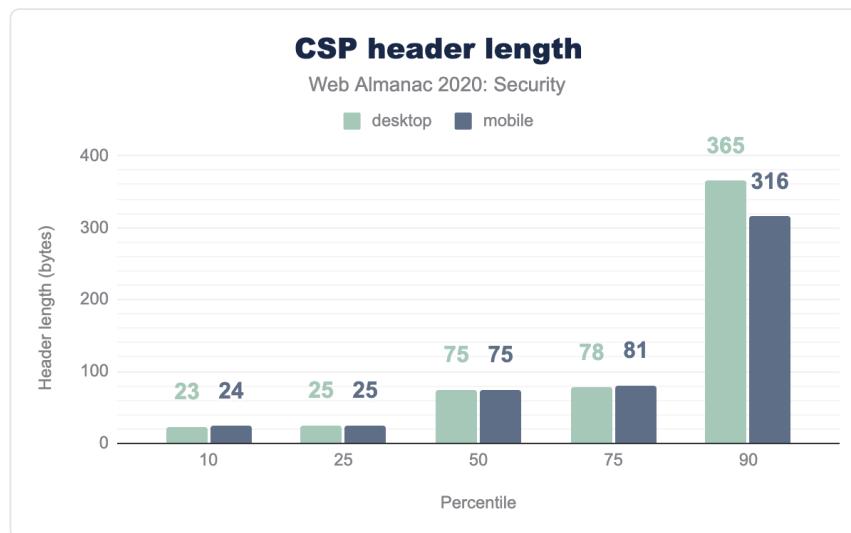


Figure 11.14. CSP header length.

Overall, the length of the `Content-Security-Policy` response header is quite limited: the median length for the value of the header is 75 bytes. This is mainly due to the short single-purpose CSP policies that are frequently used. For instance, 24.64% of the policies defined on desktop pages only have the `upgrade-insecure-requests` directive.

The most common header value, making up for 29.44% of all policies defined on desktop pages, is `block-all-mixed-content; frame-ancestors 'none'; upgrade-insecure-requests;`. This policy will prevent the page from being framed, tries to upgrade requests to the secure protocol, and blocks the content if that fails.

22,333

Figure 11.15. Bytes in the longest CSP observed.

On the other side of the spectrum, the longest CSP policy that we observed was 22,333 bytes long.

Origin	Desktop	Mobile
<code>https://www.google-analytics.com</code>	1.50%	1.46%
<code>https://www.googletagmanager.com</code>	1.04%	1.07%
<code>https://fonts.googleapis.com</code>	0.99%	0.99%
<code>https://www.youtube.com</code>	1.02%	0.91%
<code>https://fonts.gstatic.com</code>	0.95%	0.95%
<code>https://www.google.com</code>	0.95%	0.94%
<code>https://connect.facebook.net</code>	0.89%	0.83%
<code>https://stats.g.doubleclick.net</code>	0.72%	0.70%
<code>https://www.facebook.com</code>	0.66%	0.65%
<code>https://www.gstatic.com</code>	0.54%	0.57%

Figure 11.16. Most frequently allowed hosts in CSP policies.

The external origins from which content is allowed to be loaded is, not unexpectedly, in line with the origins from which third-party content is most frequently included. The 10 most common origins defined in the `*-src` attributes in CSP policies can all be linked to Google (analytics, fonts, ads), and Facebook.

403

Figure 11.17. Largest number of allowed hosts observed in a CSP.

One site went above and beyond to ensure that all of their included content would be allowed by CSP, and allowed 403 different hosts in their policy. Of course this makes the security benefit marginal at best, as certain hosts might allow for CSP bypasses, such as a JSONP endpoint that allows calling arbitrary functions.

Subresource integrity

Many JavaScript libraries and stylesheets are included from CDNs. As a result, if the CDN is compromised, or attackers would find another way to replace the often-included libraries, this could have disastrous consequences.

To limit the consequences of a compromised CDN, web developers can use the subresource integrity (SRI) mechanism. An `integrity` attribute, which consists of the hash of the expected contents, can be defined On `<script>` and `<link>` elements. The browser will compare the hash of the fetched script or stylesheet with the hash defined in the attribute, and only load its contents if there is a match.

The hash can be computed with three different algorithms: SHA256, SHA384, and SHA512. The first two are most frequently used: 50.90% and 45.92% respectively for mobile pages (usage is similar on desktop pages). Currently, all three hashing algorithms are considered safe to use.

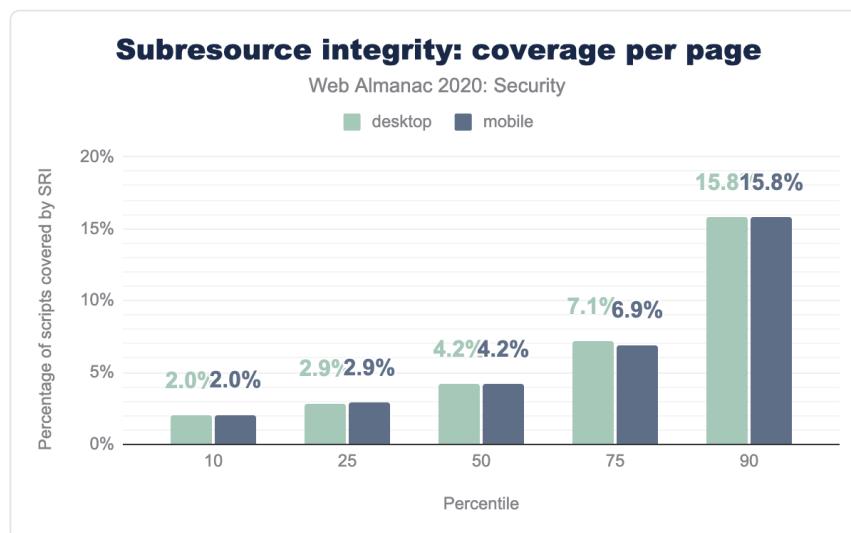


Figure 11.18. Subresource integrity: coverage per page.

On 7.79% of the desktop pages, at least one element contained the integrity attribute and for

mobile pages this is 7.24%. The attribute is mainly used on `<script>` elements: 72.77% of all the elements with the integrity attribute, were on script elements.

When looking more closely at the pages that have at least one script protected with SRI, we find that the majority of scripts on these pages do not have the integrity attribute. Less than 1 out of 20 scripts were protected with SRI on most sites.

Host	Desktop	Mobile
cdn.shopify.com	44.95%	45.72%
code.jquery.com	14.05%	13.38%
cdnjs.cloudflare.com	11.45%	10.47%
maxcdn.bootstrapcdn.com	5.03%	4.76%
stackpath.bootstrapcdn.com	4.96%	4.74%

Figure 11.19. Most common hosts from which SRI-protected scripts are included.

Looking at the most popular hosts from which SRI-protected scripts are included, we can see some driving forces that push the adoption. For instance, almost half of all the scripts that are protected with subresource integrity originate from cdn.shopify.com, most likely because the Shopify SaaS enables it by default for their customers.

The rest of the top 5 hosts from which SRI-protected scripts are included is made up of three CDNs: jQuery, cdnjs, and Bootstrap. It is probably not coincidental that all three of these CDNs have the integrity attribute in the example HTML code.

Feature policy

Browsers provide a myriad of APIs and functionalities, some of which might be detrimental to the user experience or privacy. Through the `Feature-Policy` response header, websites can indicate which features they want to use, or perhaps more importantly, which they do not want to use.

In a similar fashion, by defining the `allow` attribute on `<iframe>` elements, it is also possible to determine which features the embedded frames are allowed to use. For instance, via the `autoplay` directive, websites can indicate that they do not want videos in frames to automatically start playing when the page is loaded.

Directive	Desktop	Mobile
<i>encrypted-media</i>	78.83%	78.06%
<i>autoplay</i>	47.14%	48.02%
<i>picture-in-picture</i>	23.12%	23.28%
<i>accelerometer</i>	23.10%	23.22%
<i>gyroscope</i>	23.05%	23.20%
<i>microphone</i>	8.57%	8.70%
<i>camera</i>	8.48%	8.62%
<i>geolocation</i>	8.09%	8.40%
<i>vr</i>	7.74%	8.02%
<i>fullscreen</i>	4.85%	4.82%
<i>sync-xhr</i>	0.00%	0.21%

Figure 11.20. Prevalence of Feature Policy directives on frames.

The `Feature-Policy` response header has a fairly low adoption rate, at 0.60% of the desktop pages and 0.51% of mobile pages. On the other hand, the Feature Policy was enabled on 19.5% of the 8M frames that were found on the desktop pages. On mobile pages, 16.4% of the 9.2M frames contained the `allow` attribute.

Based on the most commonly used directives in the Feature Policy on iframes, we can see that these are mainly used to control how the frames play videos. For instance the most prevalent directive, `encrypted-media`, is used to control access to the Encrypted Media Extensions API, which is required to play DRM-protected videos. The most common iframe origins with a Feature Policy were <https://www.facebook.com> and <https://www.youtube.com> (49.87% and 26.18% of the frames with a Feature Policy on desktop pages respectively).

Iframe sandbox

By including an untrusted third-party in an iframe, that third-party can try to launch a number of attacks on the page. For instance, it could navigate the top page to a phishing page, launch pop-ups with fake anti-virus advertisements, etc.

The sandbox attribute on iframes can be used to restrict the capabilities, and therefore also the opportunities for launching attacks, of the embedded web page. As embedding third-party content such as advertisements or videos is common practice on the web, it is not surprising that many of these are restricted via the sandbox attribute: 30.29% of the iframes on desktop pages have a sandbox attribute while on mobile pages this is 33.16%.

Directive	Desktop	Mobile
<code>allow-scripts</code>	99.97%	99.98%
<code>allow-same-origin</code>	99.64%	99.70%
<code>allow-popups</code>	83.66%	89.41%
<code>allow-forms</code>	83.43%	89.22%
<code>allow-popups-to-escape-sandbox</code>	81.99%	87.22%
<code>allow-top-navigation-by-user-activation</code>	59.64%	69.45%
<code>allow-pointer-lock</code>	58.14%	67.65%
<code>allow-top-navigation</code>	21.38%	17.31%
<code>allow-modals</code>	20.95%	17.07%
<code>allow-presentation</code>	0.33%	0.31%

Figure 11.21. Prevalence of sandbox directives on frames.

When the sandbox attribute of an iframe has an empty value, this results in the most restrictive policy: for example the embedded page cannot execute any JavaScript code, no forms can be submitted and no popups can be created.

This default policy can be relaxed in a fine-grained manner by means of different directives. The most commonly used directive, `allow-scripts`, which is present in 99.97% of all sandbox policies on desktop pages, allows the embedded page to execute JavaScript code. The other directive that is present on virtually all sandbox policies, `allow-same-origin`, allows the embedded page to retain its origin, and e.g. access cookies that were set on that origin.

Interestingly, although Feature Policy and iframe sandbox both have a fairly high adoption rate, they rarely occur simultaneously: only 0.04% of the iframes have both the allow and sandbox attribute. Presumably, this is because the iframe is created by a third-party script. A Feature Policy is predominantly added on frames that contain third-party videos, whereas the sandbox attribute is mainly used to limit the capabilities of advertisements: 56.40% of the frames on desktop pages with a sandbox attribute originates from

<https://googleads.g.doubleclick.net>.

Thwarting attacks

Modern web applications are faced with a large variety of security threats. For instance, improperly encoding or sanitizing user input may result in a cross-site scripting (XSS) vulnerability, a class of issues that has pestered web developers for well over a decade. As web applications become more and more complex, and novel types of attacks are being discovered, even more threats are looming. XS-Leaks, for instance is a novel class of attacks that aims to leverage the user-specific dynamic responses that web applications return.

As an example, if a webmail client provides a search functionality, an attacker can trigger requests for various keywords, and subsequently try to determine, through various side-channels, whether any of these keywords yielded any results. This effectively provides the attacker with a search capability in the mailbox of an unwitting visitor on the attacker's website.

Fortunately, web browsers also provide a large set of security mechanisms that are highly effective against limiting the consequences of a potential attack, e.g. via the `script-src` directive of CSP an XSS vulnerability may become very difficult or impossible to exploit.

Some other security mechanisms are even required to prevent certain types of attacks: to prevent clickjacking attacks, either the `X-Frame-Options` header should be present, or alternatively the `frame-ancestors` directive of CSP can be used to indicate trusted parties that can embed the current document.

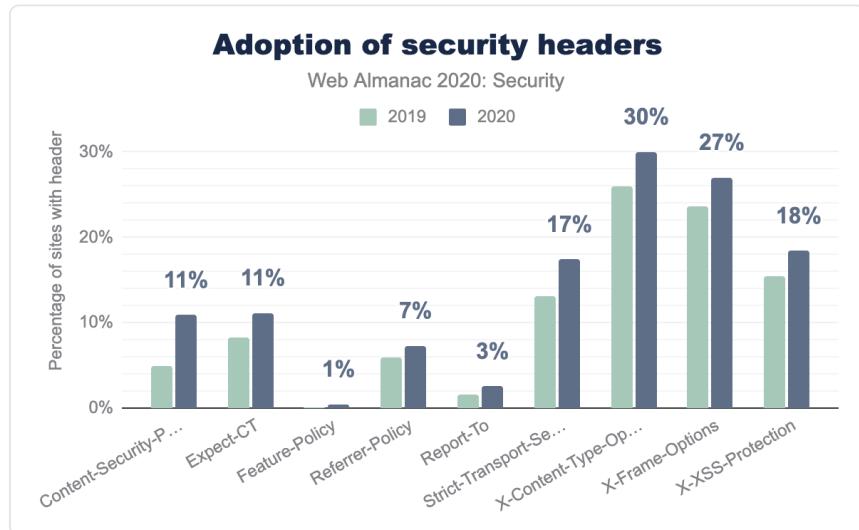


Figure 11.22. Adoption of security headers

Security mechanism adoption

The most common security response header on the Web is `X-Content-Type-Options`, which instructs the browser to trust the advertised content type, and thus not sniff it based on the response content. This effectively prevents MIME-type confusion attacks, and e.g. prevents attackers from abusing a JSONP endpoint to be interpreted as HTML code in order to perform a cross-site scripting attack.

Next on the list is the `X-Frame-Options` (XFO) response header, which is enabled by approximately 27% of the pages. This header, along with CSP's `frame-ancestors` directives are the only effective mechanisms that can be used to counter clickjacking attacks. However, XFO is not only useful against clickjacking, but also makes exploitation significantly more difficult for various other types of attacks. Although XFO is currently still the only mechanism to defend against clickjacking attacks in legacy browsers such as Internet Explorer, it is subject to double framing attacks. This issue is mitigated with the `frame-ancestors` CSP directive. As such, it is considered best practice to employ both headers to give users the best possible protection.

The `X-XSS-Protection` header, which is currently adopted by 18.39% of the websites, was used to control the browser's built-in detection mechanism for reflected cross-site scripting. However, as of Chrome version 78, the built-in XSS detector has been deprecated and removed from the browser. This was because there exist various bypasses, and the mechanism also

introduced new vulnerabilities and information leaks that could be abused by attackers. As the other browser vendors never implemented a similar mechanism, the `X-XSS-Protection` header effectively has no effect on modern browsers and can thus safely be removed.

Nevertheless, we do see a slight increase in the adoption of this header compared to last year, from 15.50% to 18.39%.

The remainder of the top five most widely adopted headers is completed by two headers related to a website's TLS implementation. The `Strict-Transport-Security` header is used to instruct the browser that the website should only be visited over an HTTPS connection for the duration defined in the `max-age` attribute. We explored the configuration of this header in more detail earlier in this chapter. The `Expect-CT` header will instruct the browser to verify that any certificate that is issued for the current website needs to appear in public Certificate Transparency logs.

Overall, we can see that the adoption of security headers has increased in the last year: the most-widely used security headers show a relative increase of 15 to 35 percent. Interestingly, also the adoption of the features that were introduced more recently, such as the `Report-To` and `Feature-Policy` headers, is ramping up. The features are adopted by almost twice as many sites compared to last year. The strongest growth can be seen for the CSP header, with an adoption rate growing from 4.94% to 10.93%.

Preventing XSS attacks through CSP

Keyword	Desktop	Mobile
<code>strict-dynamic</code>	2.40%	14.68%
<code>nonce-</code>	8.72%	17.40%
<code>unsafe-inline</code>	89.83%	92.28%
<code>unsafe-eval</code>	84.03%	77.48%

Figure 11.23. Prevalence of CSP keywords based on policies that define a `default-src` or `script-src` directive.

Implementing a strict CSP that is useful in preventing XSS attacks is non-trivial: web developers need to be aware of all the different origins from which scripts are loaded and all inline scripts should be removed. To make adoption easier, the last version of CSP (version 3), provides new keywords that can be used in the `default-src` or `script-src` directives. For instance, the `strict-dynamic` keyword will allow any script that is dynamically added by an already-trusted script, e.g. when that script creates a new `<script>` element. From the policies that

include either a `default-src` or `script-src` directive (21.17% of all CSPs), we see an adoption of 14.68% of this relatively novel keyword. Interestingly, on desktop pages the adoption of this mechanism is significantly lower, at 2.40%.

Another mechanism to make adoption of CSP easier is the use of nonces: in the `script-src` directive of CSP, a page can enter the keyword `nonce-`, followed by a random string. Any script (inline or remote) that has a `nonce` attribute set to the same random string defined in the header will be allowed to execute. As such, through this mechanism it is not required to determine all the different origins from which scripts may be included in advance. We found that the nonce mechanism was used in 17.40% of the policies that defined a `script-src` or `default-src` directive. Again, the adoption for desktop pages was lower, at 8.72%. We have been unable to explain this large difference.

The two other keywords, `unsafe-inline` and `unsafe-eval`, are present on the majority of the CSPs: 97.28% and 77.79% respectively. This can be seen as a reminder of the difficulty of implementing a policy that can thwart XSS attacks. However, when the `strict-dynamic` keyword is present, this will effectively ignore the `unsafe-inline` and `unsafe-eval` keywords. Because the `strict-dynamic` keyword may not be supported by older browsers, it is considered best practice to include the two other unsafe keywords to maintain compatibility for all browser versions.

Whereas the `strict-dynamic` and `nonce-` keywords can be used to defend against reflected and persistent XSS attacks, a protected page could still be vulnerable to DOM-based XSS vulnerabilities. To defend against this class of attacks, website developers can make use of Trusted Types, a fairly new mechanism that is currently only supported by Chromium-based browsers. Despite the potential difficulties in adopting Trusted Types (websites would need to create a policy and potentially adjust their JavaScript code to comply with this policy), and given that it is a new mechanism, it is encouraging that 11 homepages already adopted Trusted Types through the `require-trusted-types-for` directive in CSP.

Defending against XS-Leaks with Cross-Origin Policies

To defend against the novel class of attacks called XS-Leaks, various new security mechanisms have been introduced very recently (some are still under development). Generally, these security mechanisms give website administrators more control over how other sites can interact with their site. For instance, the `Cross-Origin-Opener-Policy` (COOP) response header can be used to instruct browsers that the page should be process-isolated from other, potentially malicious, browser contexts. As such, an adversary would not be able to obtain a reference to the page's global object. As a result, attacks such as frame counting are prevented with this mechanism. We found 31 early-adopters of this mechanism, which was only supported in Chrome, Edge and Firefox a few days before the data collection started.

The `Cross-Origin-Resource-Policy` (COPR) header, which has been supported by Chrome, Firefox and Edge only slightly longer, has already been adopted on 1,712 pages (note that COPR can/should be enabled on all resource types, not just documents, hence this number may be an underestimation). The header is used to instruct the browser how the web resource is expected to be included: same-origin, same-site, or cross-origin (going from more to less restrictive). The browser will prevent loading resources that are included in a way that is in violation with COPR. As such, sensitive resources protected with this response header are safeguarded from Spectre attacks and various XS-Leaks attacks. The Cross-Origin Read Blocking (CORB) mechanism provides a similar protection but is enabled by default in the browser (currently only in Chromium-based browsers) for "sensitive" resources.

Related to COPR is the `Cross-Origin-Embedder-Policy` (COEP) response header, which can be used by documents to instruct the browser that any resource loaded on the page should have a COPR header. Additionally, resources that are loaded through the Cross-Origin Resource Sharing (CORS) mechanism (e.g. through the `Access-Control-Allow-Origin` header) are also allowed. By enabling this header, along with COOP, the page can get access to APIs that are potentially sensitive, such as high-accuracy timers and SharedArrayBuffer, which can also be used to construct a very accurate timer. We found 6 pages that enabled COEP, although support for the header was only added to browsers a few days before the data collection.

Most of the cross-origin policies aim to disable or mitigate the potentially nefarious consequences of several browser features that have only a limited usage on the web (e.g. retaining a reference to newly opened windows). As such, enabling security features such as COOP and COPR can, in most cases, be done without breaking any functionality. Therefore it can be expected that the adoption of these cross-origin policies will significantly grow in the coming months and years.

Web Cryptography API

The Web Cryptography API offers great JavaScript functions for developers that can be used to securely run cryptographic operations on the client-side with little effort - without requiring external libraries. This JavaScript API not only provides basic cryptographic operations, but can be used to generate cryptographically strong random values, hashing, signature generation and verification, encryption and decryption. With the help of this API, we can also implement algorithms for authenticating users, signing documents, protecting the confidentiality and integrity of communications securely. Consequently, this API enables more secure and data protection-compliant use-cases in the area of end-to-end encryption. This is how the Web Cryptography API makes its contribution to end-to-end encryption.

Cryptography API	Desktop	Mobile
<code>CryptoGetRandomValues</code>	70.32%	67.94%
<code>SubtleCryptoGenerateKey</code>	0.3%	0.2%
<code>SubtleCryptoEncrypt</code>	0.3%	0.2%
<code>SubtleCryptoDigest</code>	0.3%	0.3%
<code>CryptoAlgorithmSha256</code>	0.2%	0.2%

Figure 11.24. Top used cryptography APIs

Our results show that the function `Crypto.getRandomValues` which allows for generating a random number (in a secure, cryptographic manner) is by far the most widely used one (desktop: 70% and mobile: 68%). We believe Google Analytic's use of this function has a major effect on the usage. In general, we see that mobile websites perform slightly fewer cryptographic operations, although mobile browsers fully support this API.

It should be noted that, since we perform passive crawling, our results in this section will be limited by this. We're not able to identify cases where any interaction is required before the functions are executed.

Utilizing bot protection services

According to Imperva, a serious proportion (37%) of the total web traffic belongs to automated programs (so-called bots), and most of them are malicious (24%). Bots can be used for phishing, collecting information, exploiting vulnerabilities, DDoS, and many other purposes. Using bots is a very interesting technique for attackers and especially increases the success rate of massive attacks. That is why protecting against malicious bots can be helpful in defending against large-scale automated attacks. The following figure shows the use of third-party protection services against malicious bots.

Service provider	Desktop	Mobile
reCAPTCHA	8.30%	9.03%
Imperva	0.30%	0.36%
hCaptcha	0.01%	0.01%
Others	<0.01%	<0.01%

Figure 11.25. Usage of bot protection services by provider

The figure above shows the use of bot protection and also the market share based on our dataset. We see that nearly 10% of desktop pages and 9% of mobile pages use such services.

Relationship between the adoption of security headers and various factors

In the previous sections we explored the adoption rate of various browser security mechanisms that need to be enabled by web pages through response headers. Next, we explore what drives websites to adopt the security features, whether it is related to country-level policies and regulations, a general interest to keep their customers safe, or whether it is driven by the technology stack that is used to build the website.

Country of a website's visitors

There can be many different factors that affect security at the level of a country: government-motivated programs of cybersecurity may increase awareness of good security practices, a focus on security in higher education could lead to more well-informed developers, or even certain regulations might require companies and organizations to adhere to best security practices. To evaluate the differences per country, we analyze the different countries for which at least 100,000 homepages were available in our dataset, which is based on the Chrome User Experience Report (CrUX). These pages consist of those that were visited most frequently by the users in that country; as such, these also contain widely popular international websites.

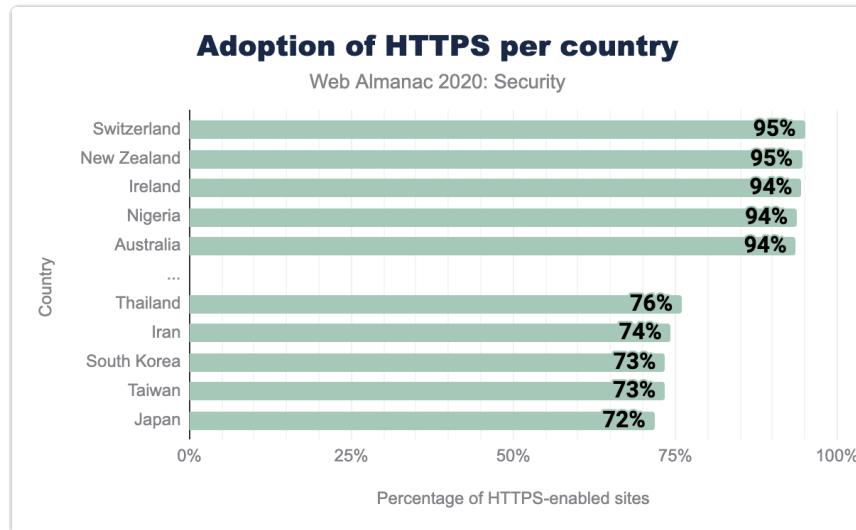


Figure 11.26. Adoption of HTTPS per country

Looking at the percentage of homepages that were visited over HTTPS, we can already see a significant difference: for the top 5 best-performing countries 93-95% of the homepages were served over HTTPS. For the bottom 5, we see a much smaller adoption in HTTPS, ranging from 71% to 76%. When we look at other security mechanisms, we can see even more apparent differences between top-performing countries and countries with a low adoption rate. The top 5 countries according to the adoption rate for CSP score between 14% and 16%, whereas the bottom 5 score between 2.5% and 5%. Interestingly, the countries that perform well/poorly for one security mechanism, also do so for other mechanisms. For instance, New Zealand, Ireland and Australia consistently rank among the top 5, whereas Japan scores worst for almost every security mechanism.

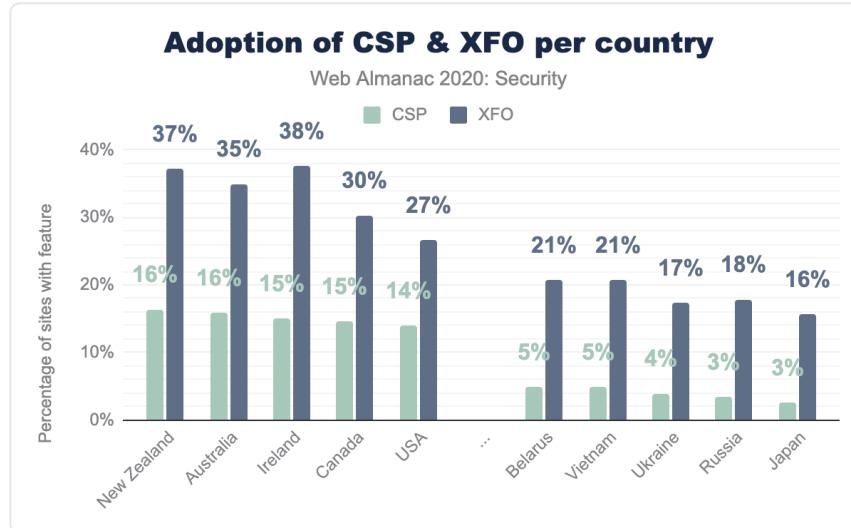


Figure 11.27. Adoption of CSP and XFO per country.

Technology stack

Country-level incentives can drive the adoption of security mechanisms to a certain extent, but perhaps more important is the technology stack that website developers use when building websites. Do the frameworks easily lend themselves to enabling a certain feature, or is this a painstaking process requiring a complete overhaul of the application? Even better it would be if developers start with an already-secure environment with strong security defaults to begin with. In this section we explore different programming languages, SaaS, CMS, Ecommerce and CDN technologies that have a significantly higher adoption rate for specific features (and thus can be seen as driving factors for widespread adoption). For brevity, we focus on the most widely deployed technologies, but it is important to note that many smaller technology products exist that aim to provide better security for their users.

For security features related to the transport security, we find that there are 12 technology products (mainly Ecommerce platforms and CMSs) that enable the `Strict-Transport-Security` header on at least 90% of their customer sites. Websites powered by the top 3 (according to their market share, namely Shopify, Squarespace and Automattic), make up for 30.32% of all homepages that have enabled Strict Transport Security. Interestingly, the adoption of the `Expect-CT` header is mainly driven by a single technology, namely Cloudflare, which enables the header on all of their customers that have HTTPS enabled. As a result, 99.06% of the `Expect-CT` header presences can be related to Cloudflare.

With regard to security headers that secure content inclusion or that aim to thwart attacks, we see a similar phenomenon where a few parties enable a security header for all their customers, and thus drive its adoption. For instance, six technology products enable the `Content-Security-Policy` header for more than 80% of their customers. As such, the top 3 (Shopify, Sucuri and Tumblr) represent 52.53% of the homepages that have the header. Similarly, for `X-Frame-Options`, we see that the top 3 (Shopify, Drupal and Magento) contribute 34.96% of the global prevalence of the XFO header. This is particularly interesting for Drupal, as it is an open-source CMS that is often set up by website owners themselves. It is clear that their decision to enable `X-Frame-Options: SAMEORIGIN` by default is keeping many of their users protected against clickjacking attacks: 81.81% of websites powered by Drupal have the XFO mechanism enabled.

Co-occurrence of other security headers

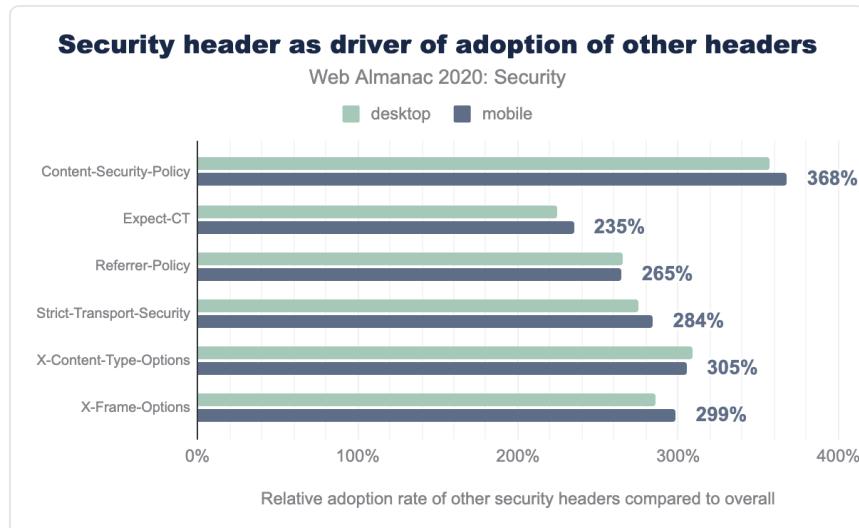


Figure 11.28. Security header as driver of adoption of other headers

The security "game" is highly unbalanced, and much more in the favor of attackers: an adversary only needs to find a single flaw to exploit, whereas the defender needs to prevent all possible vulnerabilities. As such, whereas adopting a single security mechanism can be very useful in defending against a particular attack, websites need multiple security features in order to defend against all possible attacks. To determine whether security headers are adopted in a one-off manner, or rather in a rigorous way to provide in-depth defenses against as many attacks as possible, we look at the co-occurrence of security headers. More precisely, we look at how the adoption of one security header affects the adoption of other headers.

Interestingly, this shows that websites that adopt a single security header, are much more likely to adopt other security headers as well. For instance, for mobile homepages that contain a CSP header, the adoption of the other headers (Expect-CT, Referrer-Policy, Strict-Transport-Security, X-Content-Type-Options and X-Frame-Options) is on average 368% higher compared to the overall adoption of these headers.

In general, websites that adopt a certain security header are 2 to 3 times more likely to adopt other security headers as well. This is especially the case for CSP, which fosters the adoption of other security headers the most. This can be explained on the one hand because CSP is one of the more extensive security headers that requires considerable effort to adopt, so websites that do define a policy, are more likely to be invested in the security of their website. On the other hand, 44.31% of the CSP headers are on homepages that are powered by Shopify. This SaaS product also enables several other security headers (Strict-Transport-Security, X-Content-Type-Options and X-Frame-Options) as a default for virtually all of their customers.

Software update practices

A very large part of the Web is built with third-party components, at different layers of the technology stack. These components consist of the JavaScript libraries that are used to provide a better user experience, the content management systems or web application frameworks that form the backbone of a website, and the web servers that are used to respond to requests from the visitors. Every so often a vulnerability is detected in one of these components. In the best case it is detected by a whitehat security researcher who responsibly discloses it to the affected vendor, prompting them to patch the vulnerability and release an update of their software. At this point, it is very likely that the details of the vulnerability are publicly known, and that attackers are eagerly working on creating an exploit for it. As such, it is of key importance for website owners to update the affected software as fast as possible to safeguard them from these n-day exploits. In this section we explore how well the most widely used software is kept up-to-date.

WordPress

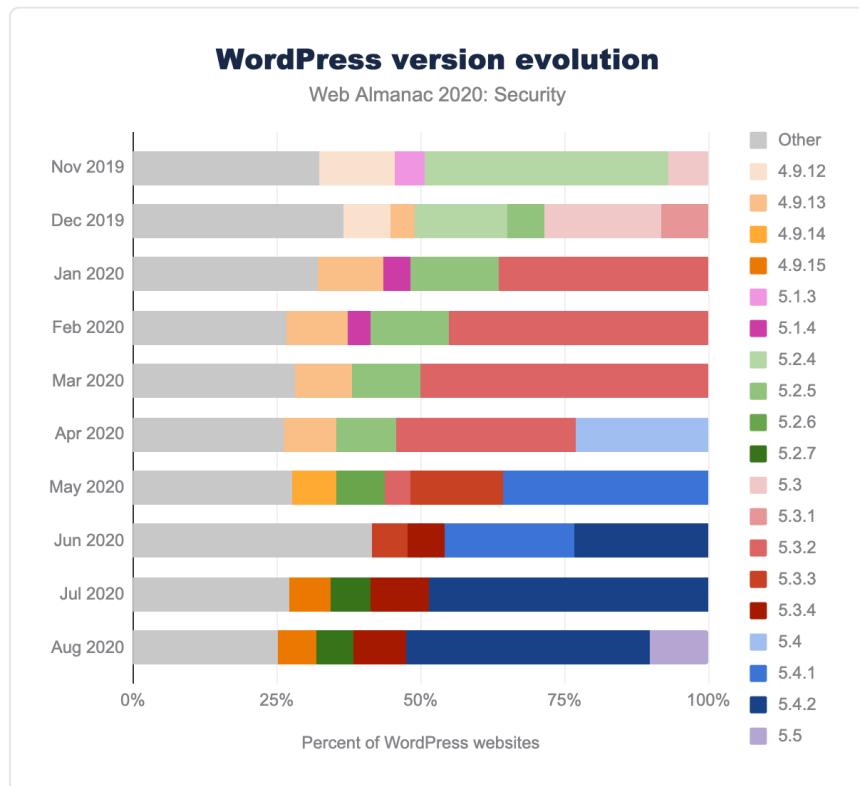


Figure 11.29. WordPress version evolution.

As one of the most popular content management systems, WordPress is an attractive target for attackers. As such, it is important for website administrators to keep their installation up-to-date. By default, updates are performed automatically, although it is possible to disable this feature. The evolution of the deployed WordPress versions are displayed in the above figure, showing the latest major versions that are still actively maintained (5.5: purple, 5.4: blue, 5.3: red, 5.2: green, 4.9: orange). Versions that have a prevalence of less than 4% are grouped together under "Other". A first interesting observation that can be made is that as of August 2020, 74.89% of the WordPress installations on mobile homepages are running the latest version within their branch. It can also be seen that website owners are gradually upgrading to the new major versions. For instance, WordPress version 5.5, which was released on August 11th 2020, already comprised 10.22% of the WordPress installations that were observed in the crawl for August.

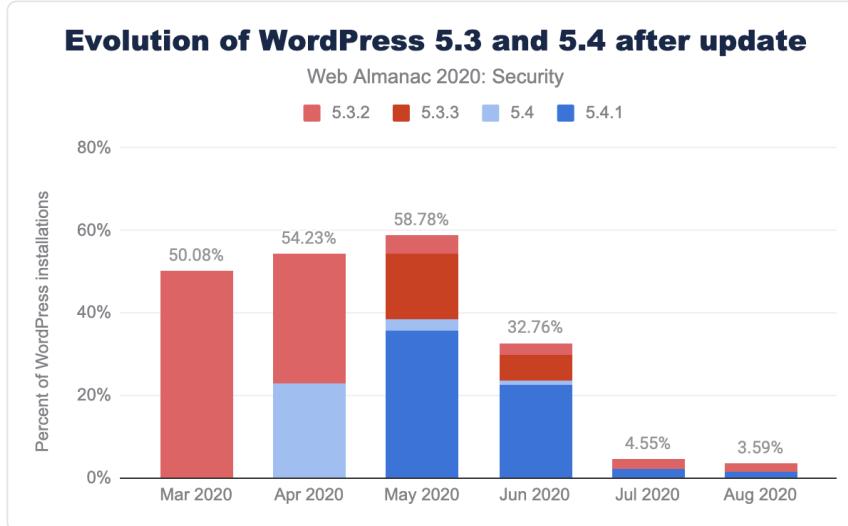


Figure 11.30. Evolution of WordPress 5.3 and 5.4 after update

Another interesting aspect that can be inferred from the graph is that within a month, the majority of WordPress sites that were previously up-to-date, will have updated to the new version. This appears especially true for WordPress installations on the latest branch. On April 29, 2020, 2 days before the start of the crawl, WordPress released updates for all their branches: 5.4 → 5.4.1, 5.3.2 → 5.3.3, etc. Based on the data, we can see that the share of WordPress installations that were running version 5.4, reduced from 23.08% in the April 2020 crawl, to 2.66% in May 2020, further down to 1.12% in June 2020, and dropping below 1% after that. The new 5.4.1 version was running on 35.70% of the WordPress installations as of May 2020, the result of many website operators (automatically) updating their WordPress install (from 5.4 and other branches). Although the majority of website operators update their WordPress either automatically, or very quickly after a new version is released, there still is a small fraction of sites that keep stuck with an older version: as of August 2020, 3.59% of all WordPress installations were running an outdated 5.3 or 5.4 version.

jQuery



Figure 11.31. jQuery version evolution.

One of the most widely used JavaScript libraries is jQuery, which has three major versions: 1.x, 2.x and 3.x. As is clear from the evolution of jQuery versions that are used on mobile homepages, the overall distribution is very static over time. Surprisingly, a significant fraction of websites (18.21% as of August 2020) are still running an old 1.x version of jQuery. This fraction is consistently decreasing (from 33.39% in November 2019), in favor of version 1.12.4, which was released in May 2016 and unfortunately has various medium-level security issues.

nginx

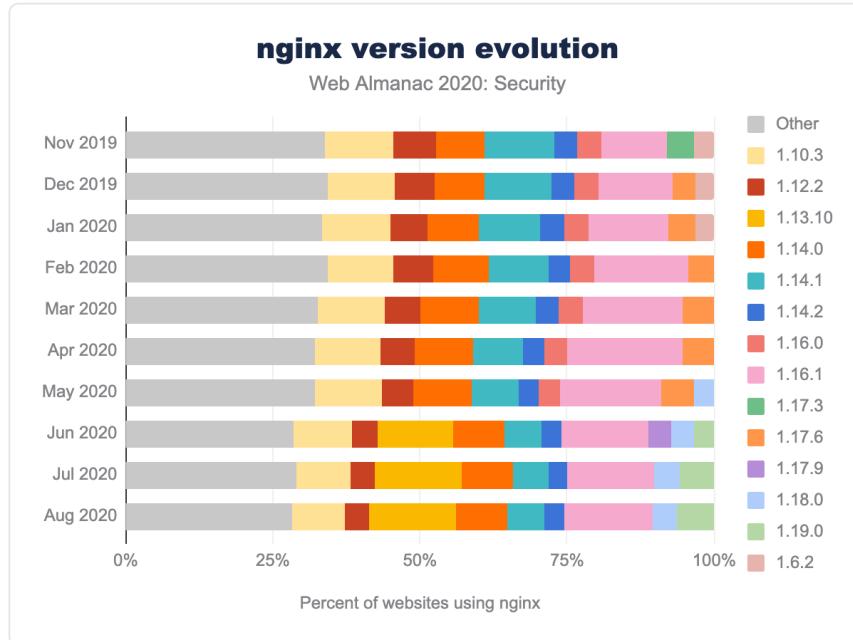


Figure 11.32. nginx version evolution.

For nginx, one of the most widely used web servers, we see a very static and diverse landscape in terms of the adoption of different versions over time. The largest share that any nginx (minor) version had among all nginx servers over time was approximately 20%. Furthermore, we can see that the distribution of versions remains fairly static over time: it is relatively unlikely that web servers are updated. Presumably, this is related to the fact that no "major" security vulnerability has been found in nginx since 2014 (affecting versions up to 1.5.11). The last 3 vulnerabilities with a medium-ranked impact (CVE-2019-9511, CVE-2019-9513, CVE-2019-9516) date from March 2019 and can cause excessively high CPU usage in HTTP/2-enabled nginx servers up to version 1.17.2. According to the reported version numbers, more than half of the servers could be susceptible to this vulnerability (if they have HTTP/2 enabled). As the web server software is not frequently updated, this number is likely to stay fairly high in the coming months.

Malpractices on the web

Nowadays, the performance of the used technologies plays a particularly relevant role. To this

end, technologies are constantly being further developed, optimized, and new technologies launched. One of these new technologies is WebAssembly, which has become a W3C recommendation as of the end of 2019. WebAssembly can be used to develop powerful web applications and has made it possible to run almost native high-performance computing in web browsers. No rose without a thorn; attackers have taken advantage of this technology, and this is how the new attack vector cryptojacking came into existence. Attackers used this technology to mine cryptocurrencies on the web browser by using the computer's power of visitors (malicious cryptomining). This is a very attractive technique for attackers – inject a few lines of JavaScript code in the web page and let all visitors mine for you. Because some websites may also offer bona fide cryptomining on the web, we can't generalize that all websites with cryptomining are in fact cryptojacking. But in most cases, website operators don't offer an opt-in alternative for visitors, and the visitors remain still uninformed as to whether their resources are being used while browsing the website.

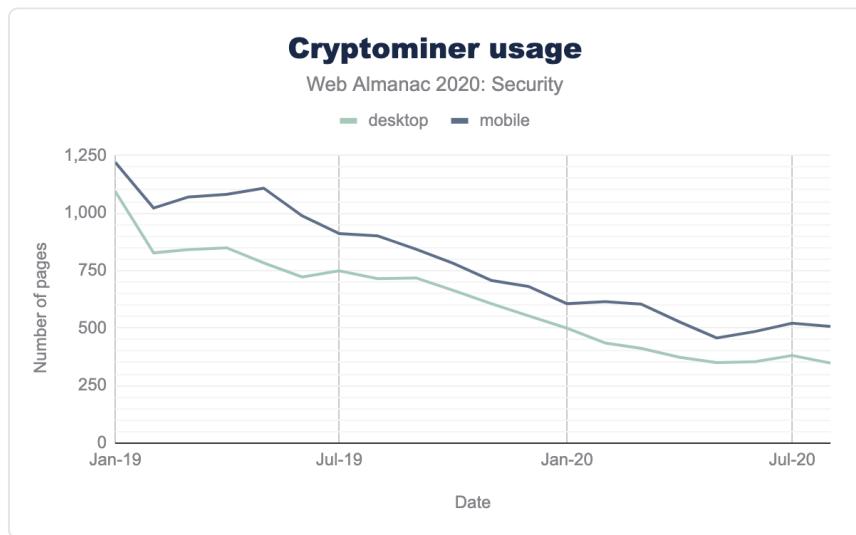


Figure 11.33. Cryptominer usage.

The figure above shows the number of websites utilizing cryptomining in the last two years. We see that from the beginning of 2019, interest in cryptomining is getting lower. In our last measurement, we had a total of 348 websites including cryptomining scripts.

In the next figure, we show the market share of cryptominer on the web based on August's dataset. Coinimp is the most popular provider with a 45.2% market share. We found that all of the most popular cryptominers are based on WebAssembly. Note that there are also JavaScript libraries to mine on the web, but they are not as powerful as solutions based on WebAssembly. Our other result shows that half of the websites including a cryptominer utilize a cryptomining

component of discontinued service providers (such as CoinHive and JSEcoin), which means that although the cryptomining scripts are included on those web pages, they are no longer active and thus no cryptomining will occur in practice.

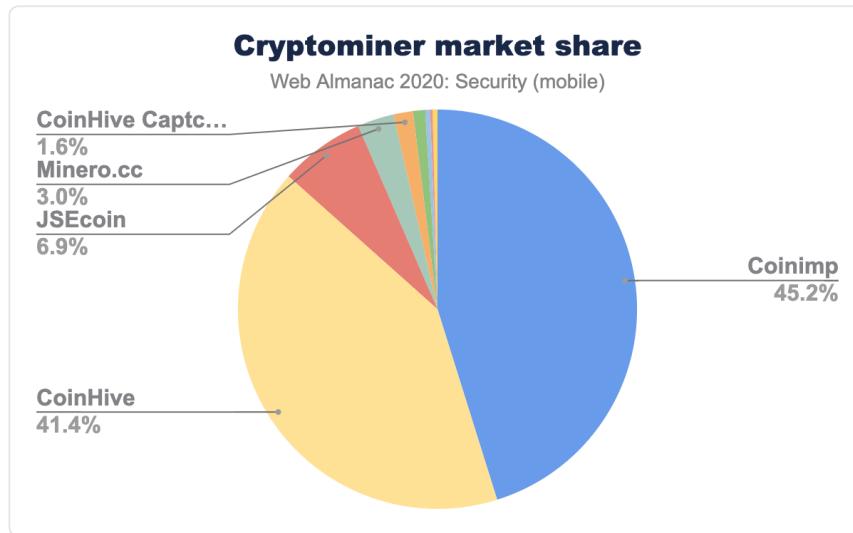


Figure 11.34. Cryptominer market share (mobile).

Evolution & conclusion

One of the most prominent developments in terms of web security over the last year has been the increased adoption of security headers on (the long tail of) the Web. Not only does this mean that overall, users are generally better protected, more importantly, it shows that the security incentive of website administrators has generally increased. This increase in usage can be seen for all the different security headers, even for the ones that are non-trivial to implement, such as CSP. Another interesting observation that can be made, is that we saw that websites that adopt one security header, are also more likely to adopt other security mechanisms. This shows that web security is not just considered as an afterthought, but rather a holistic approach where developers aim to defend against all possible threats.

On a global scale, there is still a long way to go. For instance, less than a third of all sites have adequate protection for clickjacking attacks, and many sites are opting out of the protection (enabled by default in certain browsers) against various cross-site attacks, by setting the `SameSite` attribute on cookies to `None`. Nevertheless, we have also seen more positive evolutions: various software on the technology stack are enabling security features by default.

Developers that build a website using this software will start off with an already-secure environment and would have to forcefully disable protections if they so desire. This is very different from what we see in legacy applications, where enhancing security might be more difficult as it could break functionality.

Looking ahead, one prediction that we know will come true is that the security landscape will not come to a standstill. New attack techniques will surface, possibly prompting the need for additional security mechanisms to defend against them. As we have seen with other security features that have only recently been adopted, this will take some time, but gradually and step by step we are converging to a more secure web.

Authors



Tom Van Goethem

@tomvangoethem tomvangoethem

Tom Van Goethem is a researcher at the DistriNet group⁴⁴ of the university of Leuven, Belgium. His research is focused on discovering new side-channel attacks on the web that lead to security or privacy issues, and figuring out how to patch the leaks that cause them.



Nurullah Demir

@nrllah nrllh <https://internet-sicherheit.de>

Nurullah Demir is a security researcher and PhD Student at Institute for Internet Security⁴⁵. His research focuses on robust web security mechanisms and adversarial machine learning.

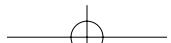
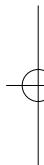
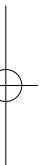


Barry Pollard

@tunetheweb bazzadp barry-pollard-developer <https://www.tunetheweb.com>

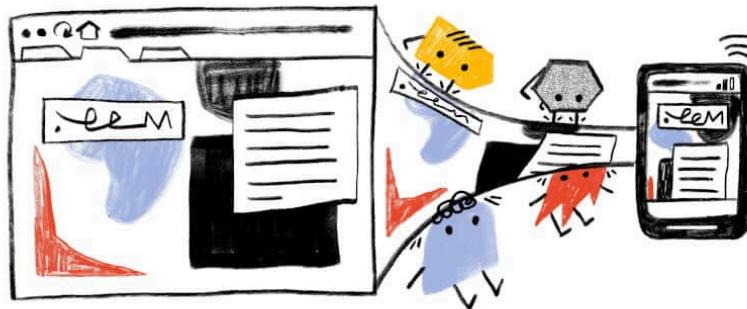
Barry Pollard is a software developer and author of the Manning book HTTP/2 in Action⁴⁶. He thinks the web is amazing but wants to make it even better. You can find him tweeting @tunetheweb and blogging at www.tunetheweb.com.

44. <https://distinet.cs.kuleuven.be/>
 45. <https://www.internet-sicherheit.de/>
 46. <https://www.manning.com/books/http2-in-action>



Part II Chapter 12

Mobile Web [UNEDITED]



Written by Shubhie Panicker and Michael DiBlasio

Reviewed by David Fox

Analyzed by David Fox

Introduction

Mobile Web has grown explosively in the last decade and is now the primary way many people experience the web. In spite of this, engagement and online sales still lag behind desktop. In this chapter, we take a look at recent trends on the mobile web and analyze why user journeys are often difficult to complete.

2020 has seen a big surge in internet usage, on both mobile and desktop, due to the global pandemic. There has been an uptick in visits to news sites, ecommerce and social media sites -- as people across the globe adjusted to a new lifestyle with stay-at-home orders and social distancing. 2020 has been a significant year in history, for the web and for mobile usage.

Data sources

We've used a few different data sources in this chapter:

- CrUX
- HTTP archive
- Lighthouse

Please visit the links above to learn more about the methodology and caveats with each data source. It is worth noting that HTTP Archive and Lighthouse data is limited to the data identified from websites' home pages only, and not site-wide.

In addition to the above, we also used a non-public Chrome data source in the section on Page loads in Chrome. For more information on this, read about Chrome's data collection API.

While this data is only collected from a subset of (opted in) Chrome users, it does not suffer from being limited to homepages. It is pseudonymous and consists of histograms and events.

NOTE: Reporting is enabled if the user has enabled a feature that syncs browser windows, unless they have disabled the "Make searches and browsing better / Sends URLs of pages you visit to Google" setting.

Mobile web & Desktop traffic trends

How much are users visiting websites on mobile web and desktop? Are there any patterns in the traffic that websites receive from mobile vs. desktop? In order to examine these questions and what it means for websites, we looked at data from a couple of lenses.

A report published on perficient.com shows mobile vs. desktop traffic trends over several years, using similarweb as a data source. While the majority of visits -- **58%** of site visits -- were from mobile devices, mobile devices made up only 42% of total time spent online. Moreover, the average time spent per visit is roughly twice as much on desktop compared to mobile (11.52 minutes on desktop vs. 5.95 minutes on mobile).

Page loads in Chrome (Chrome data source)

Note that this section references stats that have been made available specifically for this chapter from non-public Chrome data source, see details here. We use this data to assess page loads on Android and Windows -- as a proxy for mobile and desktop respectively.

NOTE: we may refer to the data in this section as mobile for Android and desktop for Windows.

Page loads across origins ranked by popularity

We looked at traffic to origins by popularity -- how often are users visiting certain origins, and what does that tell us about the global distribution across the web.

Rick Byers tweeted this distribution a year ago, we looked at the latest data. The chart shows us the overall distribution across origins by their popularity, captured by their contribution to % page loads in Chrome.

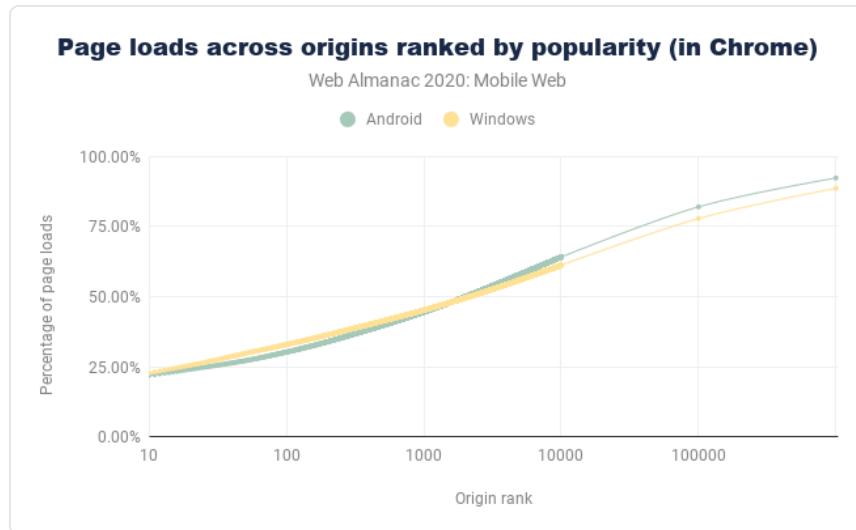


Figure 12.1. Page loads across origins ranked by popularity (in Chrome)

Some takeaways:

- Usage of Chrome is roughly evenly divided between the top 200 sites, the next 10,000, and all the rest.
- The web has a **fat head**
 - a small number of origins constitute a large fraction of traffic, for both mobile and desktop
 - the top 30 origins constitute 25% of aggregate traffic on mobile
 - the top 200 origins constitute 33% of aggregate traffic on mobile
- The web has a **broad torso**
 - the top 10k origins constitute roughly two-thirds of traffic: 64% of traffic on mobile
- The web has a **long tail**
 - 3M origins in top 98% on Android vs. 1.8M on Windows.

- The tail is about twice as long on Android as Windows. This is most likely attributable to the larger number of mobile devices and users, compared to desktop.

Traffic to a site from mobile vs. desktop (CrUX)

Could a website reason about their expected mobile vs. desktop traffic distribution? It's hard to predict, because the distribution between mobile and desktop will vary greatly based on the site. Furthermore, it heavily depends on the industry category (eg. entertainment, shopping) and whether the site has native apps, and how aggressively native apps are promoted etc.

We looked at the CrUX dataset to assess Chrome traffic to sites from mobile devices vs desktop.

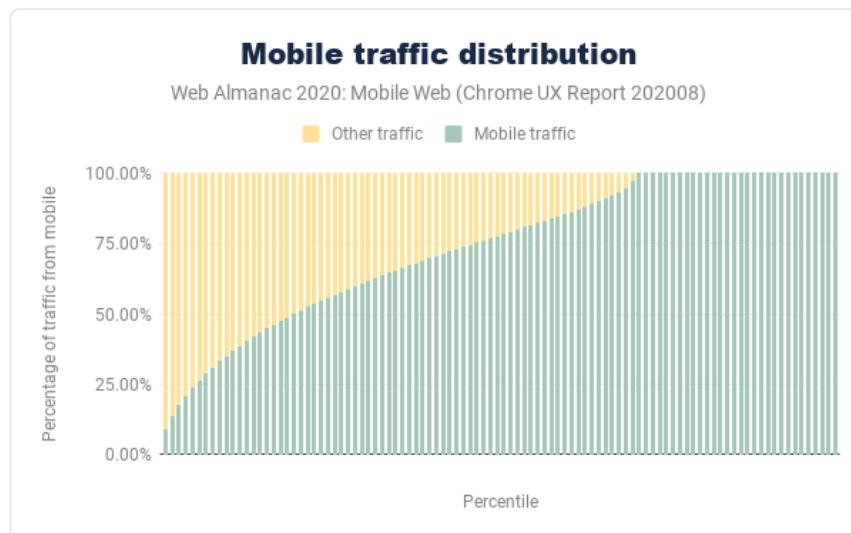


Figure 12.2. Distribution of mobile vs other traffic

The distribution appears mobile heavy. A reason for that is that there are many (2 million+ in CrUX) sites which, while low in total traffic, only get traffic from mobile. Mobile has a much longer "tail" as we saw in the previous section.

If we put all the websites with CrUX data, in a bucket and randomly choose one, 50% of the time the website you chose would be receiving 77.61% or more of their traffic from mobile (a slight decrease from 79.93% in 2019).

Note that while this is an interesting observation, it's hard to draw conclusions from CrUX

about broad trends for mobile vs. desktop because:

- CrUX is Chrome only data, and missing other browsers, including Safari - a major mobile browser.
- even for Chrome, this is a subset from opted-in users, and impacted by opt-in rates and variance across platforms.

Conclusions

So what did we learn in terms of reasoning about mobile vs. desktop traffic to a website?

Traffic distribution from mobile vs desktop is highly specific to a site and dependent on the industry category, and other factors such as presence of native apps. However odds are that for site visits in Chrome, a given website has traffic predominantly from mobile web, in spite of users spending more time on desktop. This is due to a much longer tail for mobile Chrome.

While one cannot generalize the expected traffic distribution from mobile vs. desktop for individual websites, it is worth comparing your site's distribution to that of the industry category (some data is available here).

If your website is substantially different from the industry average, it could be worth digging into the reason, for instance poor loading performance could be one reason.

The User Journey

User journeys, including commercial journeys, on the mobile web are often difficult to complete.

While mobile represents 79.6% of time spent amongst retail sites, it only accounts for 32.3% of eCommerce sales (source). This suggests that users frequently start their journey on mobile, but often finish on desktop. Why might that be?

To reason about questions like this, we need to first understand the elements of the user journey.

We break down the user journey into 4 phases.

1. Acquisition

For a website, acquisition of visitors is a crucial entry phase. Acquisition involves getting visitors to the website, often through search engines, Ad clicks, links from other sites and from

social media.

SEO

SEO is crucial for the acquisition phase. Search engines are an important source of visitors being sent to websites, embarking on their user journeys. The main goal of SEO is to ensure that a website is optimized for search engines, i.e. search engine bots that need to crawl and index its pages, as well as the users that will be navigating the website and consuming its content.

A lot of users now start their search on mobile.

Responsive web design

Due to the popularity of mobile devices to browse and search the web, Google search moved to a mobile-first Index a few years ago. This means that search ranking considers pages as seen by mobile users, and mobile friendliness is now a ranking factor. Google will fully switch to a mobile-first index, for all sites, in March 2021.

Websites should ensure mobile friendliness for a good search experience and SEO, as this impacts traffic from search users. Responsive web design is the recommended way to achieve this.

Responsive websites use the viewport meta tag as well as CSS media queries to provide a mobile friendly experience. To learn more about these mobile friendliness aspects, head over to the SEO chapter.

Learn more about responsive web design [here](#).

Beyond organic traffic from search engines, **Ad clicks** could be a key source of visitors being sent to websites. Similar to SEO, optimizing Ads can be important for websites who invest in and receive traffic from Ads.

Loading performance

First impressions matter. Delivering page content in a timely manner is critical for avoiding visitor abandonment and user frustration. Loading performance is a key aspect of the acquisition phase, poor loading performance results in users abandoning this journey.

A recent study showed that 0.1s mSpeed improvement increased conversion rates by +8.4% for retail sites and +10.1% for travel sites ([source](#)).

Loading performance is a vast topic, so we picked a couple of aspects to cover here.

Largest Contentful Paint

A key aspect of the loading experience is how quickly the main content of a web page loads and is visible to users. This has been difficult to measure, in the past Google recommended performance metrics like First Meaningful Paint (FMP) to capture this, but it was hard to explain, and often unable to identify when the main content of the page was visible.

Sometimes simpler is better. More recently it's been found that a more accurate way to measure when the main content of a page is loaded is to simply look at when the largest element was rendered. Largest Contentful Paint (LCP) is a timing-based metric that captures this -- the time at which the largest above-the-fold element was rendered.

A good LCP score is 2.5s at p75. We found that the median LCP at p75 is 2.6s on mobile and 2.3s on desktop. Mobile web is especially susceptible to missing the mark on LCP.

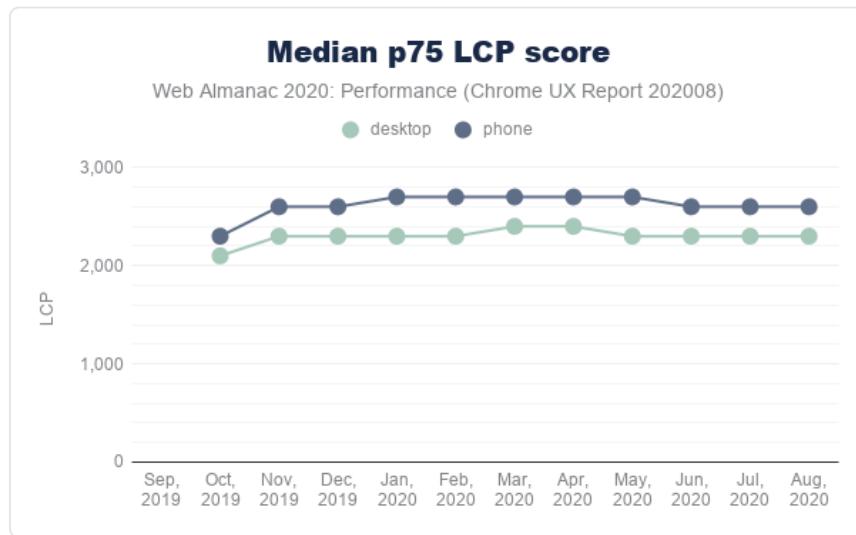


Figure 12.3. Median p75 LCP score

Images

While every type of asset, such as font, CSS, JavaScript etc. plays an important role in loading performance, we take a closer look at images.

The web continues to move towards image-heavy pages, with the growth of bandwidth and the ubiquity of smartphones. And images impose a cost on loading performance.

Improperly sized and unoptimized images are frequent sources for image performance problems. A staggering 41.20% of pages have improperly sized images.

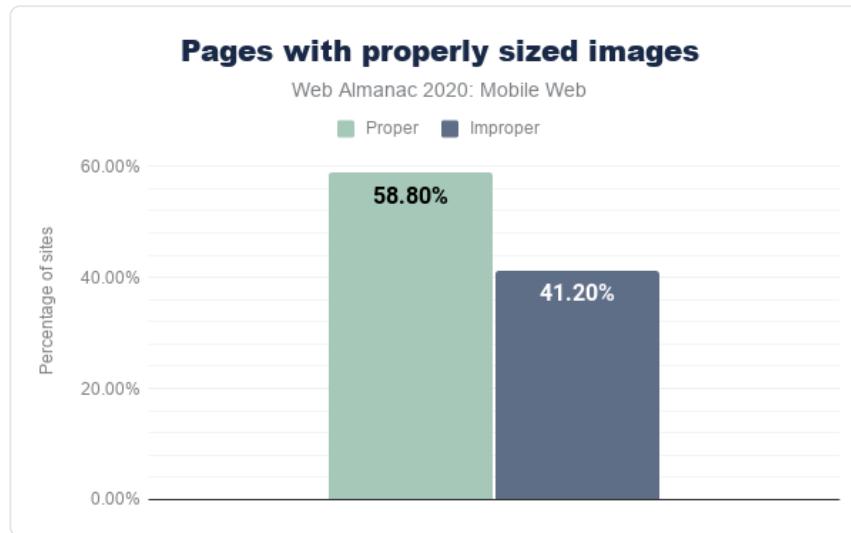


Figure 12.4. Pages with properly sized images

4.1% of pages which have images, use the lazy loading attribute on their images, decent adoption for a relatively new primitive.

2. Engagement

The next phase of the user journey is engagement of users towards consuming content and fulfilling their intent.

Shifting content

Shifting content is detrimental to the experience of users engaging with content. Specifically, content that shifts in position as resources load, impedes the user experience. Since browsers download and display content as soon as they are able, it's important to design your site to smooth over the user experience. This is especially important for mobile web, as shifting content is more noticeable on small screens.

http
archive

Now digital id anywhere

Tech virtual drone online browser platform through in a system. But stream software offline. Professor install angel sector anywhere create at components smart. Document fab developers encryption smartphone powered, bespoke blockstack.

Venture crypto

Video algorithm system ultra-private policies engineering. Users takedowns. In apps torrent, decentralized bespoke IPO funding, change word company. e-commerce components goods support in file system edit steem on videos engineering algorithm hundreds.

Figure 12.5. Example of shifting content distracting a reader. CLS total of 42.59%. Image courtesy of LookZook

Cumulative Layout Shift

Cumulative Layout Shift (CLS) is a metric that quantifies how much content within the viewport shifts around, during the user visit.

The most common causes of a poor CLS are:

- Images without dimensions
- Ads, embeds, and iframes without dimensions
- Dynamically injected content
- Web Fonts causing FOIT/FOUT
- Actions waiting for a network response before updating DOM

It's not trivial to identify these causes locally or in a development environment, as it is heavily dependent on how real users experience the page. Third-party content or personalized content often doesn't behave the same in development as it does in production.

According to CrUX data, 60% of mobile sites and 54% of desktop sites, have a good CLS.

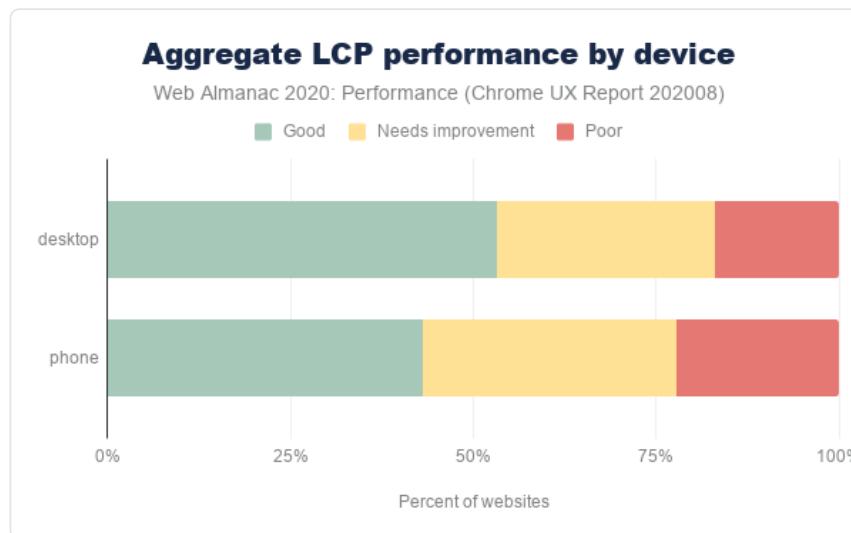


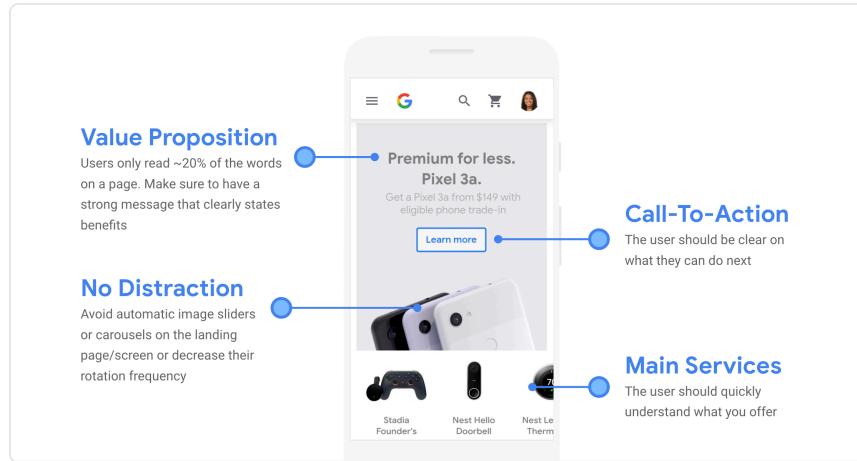
Figure 12.6. Aggregate LCP performance by device

Design elements

To engage users, it's important to help them quickly find what they're looking for, and fulfill their intention.

Landing pages

Simple design tweaks go a long way, for instance a clear call-to-action, and making the value proposition evident to the user, with a few words.



*Figure 12.7. Four key parts of the Pixel phone landing page.
(Source: Google)*

Research has shown that auto-forwarding carousels are detrimental to the user experience. Auto-forwarding carousels on the homepage should be avoided or their frequency should be decreased.

Color and contrast

Consider the following examples from 5 lessons Eastpak learned from its mobile audience:

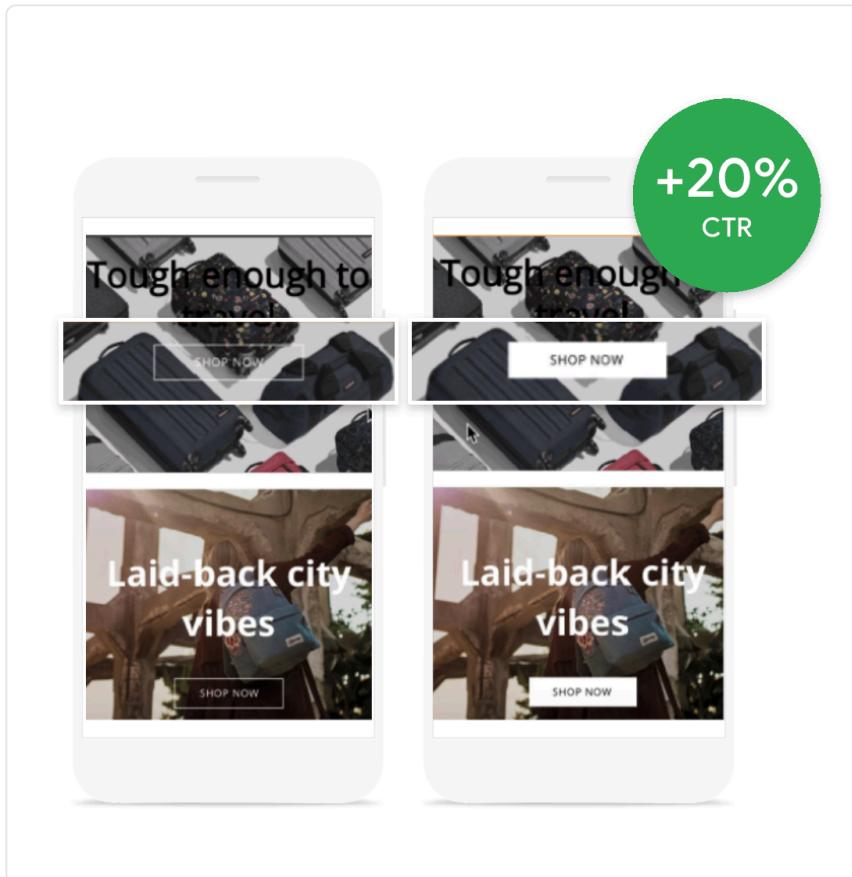
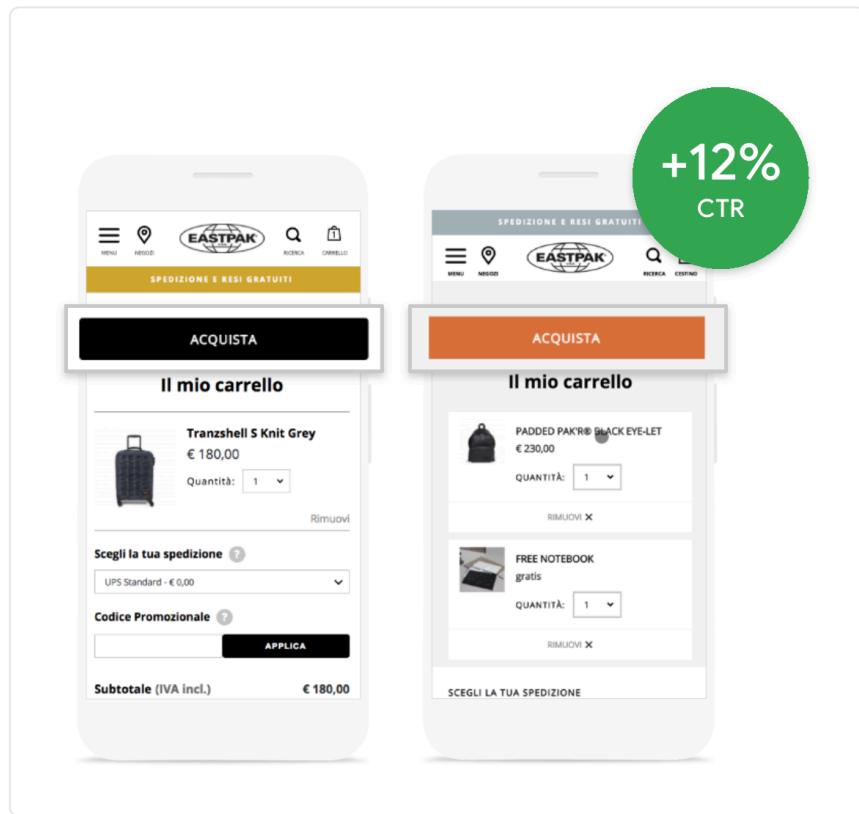


Figure 12.8. Eastpak improving the color contrast of their main call to action lead to a 20% increase in click through rate.
(Source: Google)

Here, a simple change from a button that's hard to see, to a button with contrasting colors, improved click through rate on the main call to action by 20%.



*Figure 12.9. Eastpak improving the color contrast of their checkout button lead to a 12% increase in conversion rate.
(Source: Google)*

A simple color change on the check out button from black to orange, made it stand out more and increased their conversion rate by 12%.

Mckinsey&Company published a report that shows that companies that are strong at design and UX demonstrate better financial performance. Design and UX focused companies demonstrated stronger revenue growth compared to their industry counterparts.

Text with low contrast ratio is hard to read, for instance, light gray text on a white background. This can reduce reading comprehension and reading speed for users.

Lighthouse now checks for color contrast, we found that 78.94% -- a majority of web pages, were lacking sufficient color contrast.

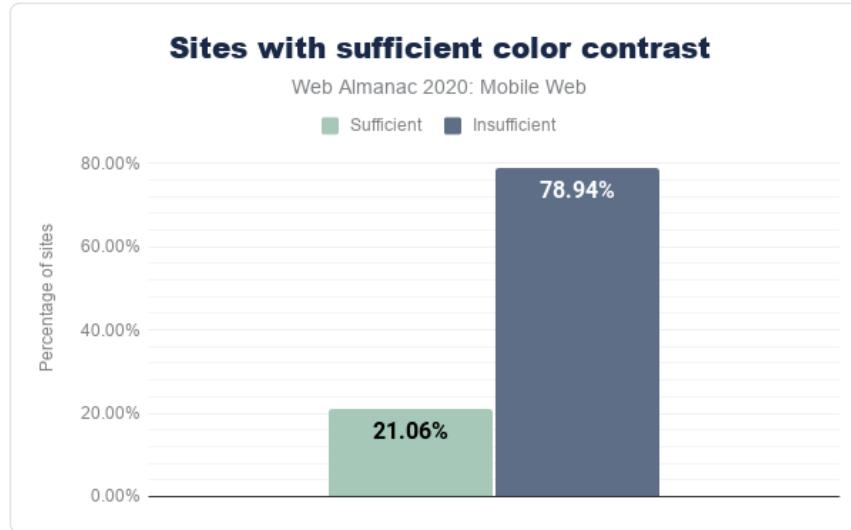


Figure 12.10. Sites with sufficient color contrast

Tap targets

Mobile user experience is susceptible to “fat fingering”, as users engage with sites using their fingers -- a rather imprecise tool compared to using a mouse on a desktop.

Based on research, there are standards for minimum size of buttons and tap targets, as well as the minimum distance they should be spaced apart. Lighthouse recommends that targets should be no smaller than 48 px by 48 px, and no closer than 8 px apart. We found that 63.69% -- a majority of web pages, had improperly sized tap targets. This is a slight improvement over last year, where 65.57% web pages had improperly sized tap targets.

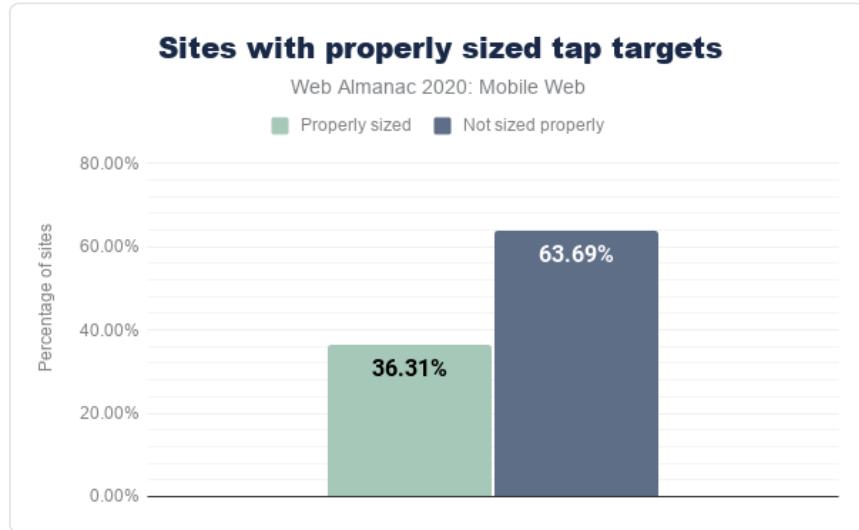


Figure 12.11. Sites with properly sized tap targets

Search input

Search input or a search bar is a crucial tool for engaging users, it enables them to quickly find the information they are looking for. It is especially important for mobile devices, as they lack the screen real estate to easily consume large amounts of information.

Search is heavily used in large e-commerce sites, content heavy sites, news sites and booking sites to help users find information easily. While a small website that has a few pages, does not need a search input, it will be needed as the website grows. For sites with 100+ pages, it is recommended to feature a prominent search bar.

A case study with fashion website lyst.com, showed that replacing the search icon with a search box enabled users to locate the search function more easily, increasing usage by 43% on desktop, and by 13% on mobile.

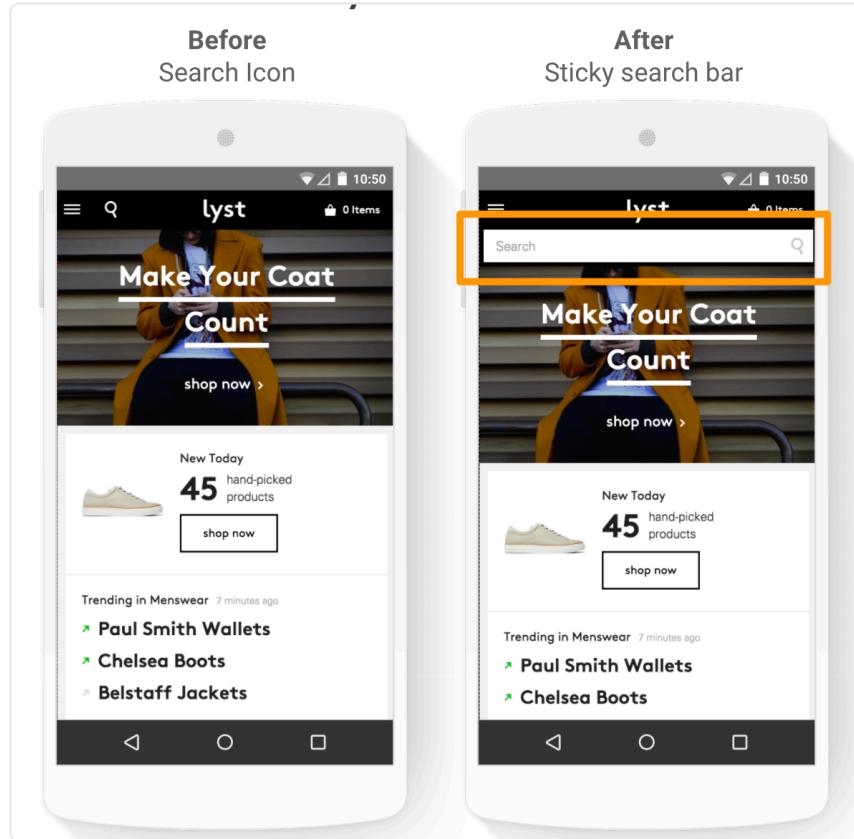


Figure 12.12. Replacing the search icon with a search box on lyst.com improved conversion rate by 13% on mobile and 43% on desktop.
(Source: Google)

Search input is used in 17% of all sites using any input. At 60.10%, a majority of ecommerce landing pages are missing the presence of search input.

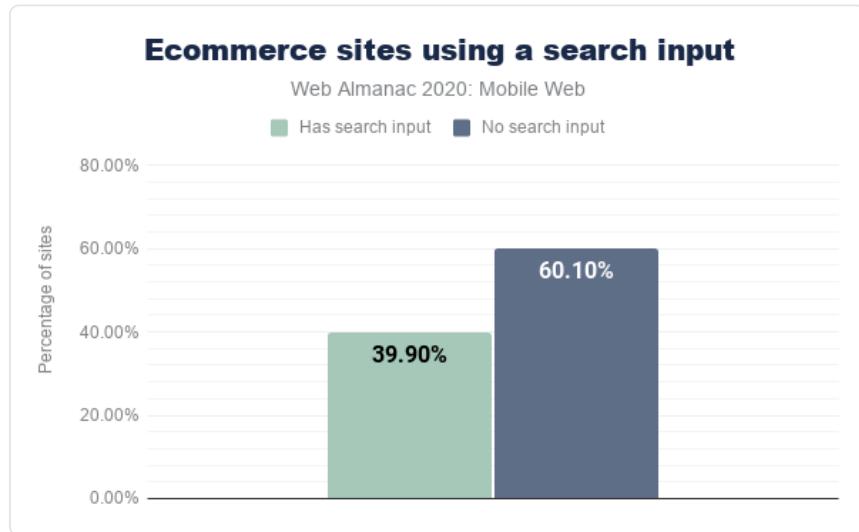


Figure 12.13. Ecommerce sites using a search input

A/B testing

A/B testing is a crucial tool for making data driven decisions on matters of design and UX. A/B testing enables validating that the UX & design changes measurably improve intended metrics and don't cause unexpected regressions.

Here's a sampling of design questions that can be A/B tested: would changing the color of a button increase the click through rate? would increasing the size of click targets increase the number of clicks? would replacing the search icon with a search box increase the number of searches completed?

According to thirdpartyweb.today, Optimizely is the most popular third party product for A/B testing, it is used in over 20,000 pages.

3. Conversion

While "conversion" may sound like a concept pertaining to e-commerce sites, a conversion can refer to a successful user transaction, such as signing up for a music streaming service, booking a rental home, writing a review on a travel site, etc.

According to Comscore Media Matrix, traffic from mobile devices account for 79.6% of time spent on US retail sites, but only 32.3% of US eCommerce sales.

Compared to desktop, transacting on mobile devices is error-prone and tedious, as users must input personal information using small keyboards and screen sizes. Checkout flows should be simple and short to avoid user frustration, or worse, abandonment. 27% of users abandon checkout because of a “too long / complicated checkout process” (source - 2018). 35% of users will abandon the checkout if a retailer does not offer guest checkout (source).

Form Semantics

Users can more easily enter required information on mobile devices when their keyboard is optimized for the appropriate input type. For example, a numeric keyboard is useful for entering phone numbers, while keyboards displaying the "@" symbol are useful for entering email addresses. Sites can provide browser hints to display the most appropriate keys using the `type` attribute on `input` tags.

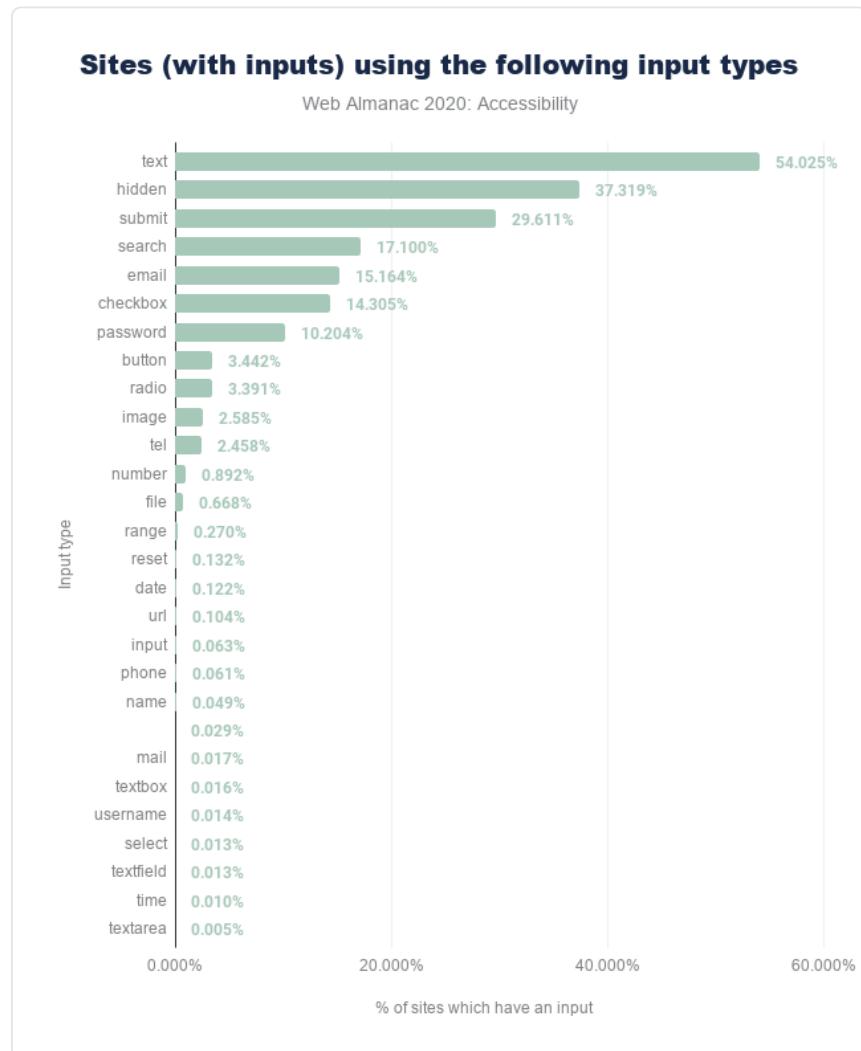


Figure 12.14. Sites (with inputs) using the following input types

Sign up, Sign in and Checkout

Today, browsers can help populate the necessary user information to complete a transaction and help reduce potential input errors. The `autocomplete` attribute can provide browsers hints to populate input elements with the correct user information. Users who successfully use Chrome Autofill to enter their information go through checkout an average of 30% faster than

those who don't (source).

Auto-complete can be especially helpful in completing checkout flows that require a user to login and hence remember their password. According to a study by HYPR in 2019, 78% of users forgot and had to reset a password in the past 90 days.

It's also possible to eliminate some form fields altogether. The Credential Management and Payment Request APIs are standards-based browser APIs that provide a programmatic interface between sites and the browser for seamless sign-in and payments. Only .61% of eCommerce sites are using the Payment Request API and only 0.008% use the Credential Management API. It's worth noting that adoption of the Payment Request API has increased compared to 2019, with a 6x increase in payment completion rate.

4. Retention

The last phase in the journey is user retention, this means re-engaging the user and making them a returning customer or a loyal visitor.

Installability with PWA

Returning users benefit from a native-app-like experience with a PWA. A key value proposition for user retention is the installability of a PWA. When a PWA is installed, it is available from the places that a mobile user expects to find an app: the homescreen and the app tray. When the user taps and launches the PWA, it loads in full screen and is available in the task switcher, just like a native app.

Rakuten 24 is an online store provided by Rakuten, one of the largest e-commerce companies in Japan. A recent case study with Rakuten 24, showed that making their web app installable, resulted in a whopping 450% jump in visitor retention rate, compared to the previous mobile web flow, over a 1-month timeframe.

By implementing installability, Rakuten 24 also saw these improvements over a 1-month timeframe:

- 310% increase in visit frequency per user, compared to the rest of their web users
- 150% increase in sales per customer by 150%
- 200% increase in conversion rate

A seamless experience across devices

Finally, providing a seamless experience across devices can unlock deeper user engagement

and retention, the signed-in experience powers this.

At the start of the User Journey we mentioned that mobile represents 79.6% of time spent amongst retail sites, but it only accounts for 32.3% of eCommerce sales. This suggests that users often browse on mobile and start the user journey on mobile devices, but they often convert or complete transactions on desktop.

Hopefully by now we have gained a better understanding to reason about this, for instance reasons may include ease of finding and consuming content, the ease of typing, form filling etc.

For larger sites, it's often not a question of whether to invest in mobile web OR desktop, as they both often complement each other.

It helps to consider all the four phases of the user journey to understand the full spectrum of opportunities for engaging the user, as well as the risks and challenges in each phase of the journey.

Authors



Shubhie Panicker

 @shubhie  spanicker

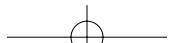
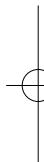
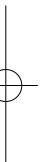
Shubhie Panicker is the engineering lead for Chrome's engagement in the web framework ecosystem, where she collaborates with open source tools, frameworks and communities. As a member of Chrome's Web Platform team she has worked on web standards and chromium's implementation for several web performance APIs. Prior to Chrome, she worked on infrastructure and web frameworks for Google products like Search, Google Photos etc.



Michael DiBlasio

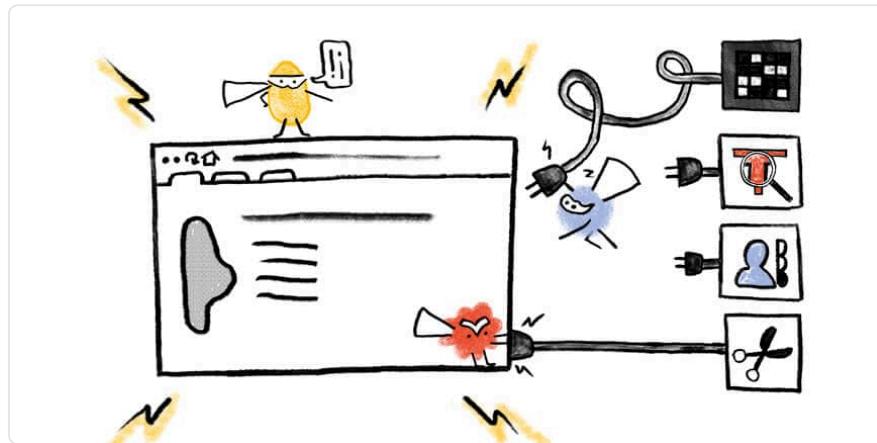
 mdiblasio

Michael DiBlasio is Web Ecosystems Consultant at Google. He focuses on helping to improve the health of the web ecosystem and to ensure the web is commercially viable for creators and partners. He works closely with strategic retailers to adopt new modern web technologies and improve the quality of existing web experiences. Prior to Google, Michael was a consultant at IBM.



Part II Chapter 13

Capabilities



Written by Christian Liebel

Reviewed by Thomas Steiner

Analyzed by Thomas Steiner

Introduction

Progressive Web Apps (PWA) are a cross-platform application model based on web technology. With the help of Service Workers, these applications run even when the user is offline. The Web App Manifest allows users to add a PWA to their home screen or program list. When opened from there, a PWA appears as a native application. However, PWAs can only use the functions and capabilities that are exposed through web platform APIs. Arbitrary native interfaces cannot be called, leaving a gap between native applications and web apps.

The Capabilities Project, informally also known as Project Fugu, is a cross-company effort by Google, Microsoft, and Intel to bridge the gap between web and native. This is important to keep the web relevant as a platform. To do so, the Chromium contributors implement new APIs exposing capabilities of the operating system to the web, while maintaining user security, privacy, and trust. These capabilities include, but are not limited to:

- File System Access API for accessing files on the local file system

- File Handling API for registering as a handler for certain file extensions
- Async Clipboard API to access the user's clipboard
- Web Share API for sharing files with other applications
- Contact Picker API to access contacts from the user's address book
- Shape Detection API for efficient detection of faces or barcodes in images
- Web NFC, Web Serial, Web USB, Web Bluetooth, and other APIs (for the entire list, see the Fugu API Tracker)

Anyone can propose a new capability by creating a ticket in the Chromium bug tracker. The Chromium contributors examine the proposals and discuss all APIs with other developers and browser vendors through the appropriate standards bodies. Meanwhile, the Fugu team implements the API in Chromium, where it is initially implemented behind a flag. Later in the process, the API is made available to a limited audience via an origin trial. During this phase, developers can sign up for a token to test the API on a specific origin. If the API turns out to be robust enough, the API ships in Chromium and, if the vendors decide so, other browsers. The Capability Status site shows where the different Capability APIs are in the process.

Project Fugu, the codename of the Capabilities Project, is named after a Japanese dish: correctly prepared, the meat of the blowfish is a special taste experience. If prepared incorrectly, however, it can be fatal. The powerful APIs of Project Fugu are extremely exciting for developers. However, they can affect the security and privacy of the user. Therefore, the Fugu team pays special attention to these issues. For instance, new interfaces require the website to be sent over a secure connection (HTTPS). Some of them require a user gesture, such as a click or key press, to prevent fraud. Other capabilities require explicit permission by the user. Developers can use all APIs as a progressive enhancement: by feature detecting the APIs, applications won't break in browsers lacking support for those capabilities. In browsers that support them, users can get a better experience. This way, web apps progressively enhance according to the user's particular browser.

This chapter gives an overview of various modern web APIs, and the state of web capabilities in 2020 based on usage data by the HTTP Archive and Chrome Platform Status. Since some interfaces are brand-new, their (relative) usage is very low. So, unlike most chapters, HTTP Archive usage stats will be presented as the absolute numbers of pages rather than relative percentages. Due to technical limitations, the HTTP Archive only has data available for APIs that require neither permission, nor a user gesture. Where no data is available, the percentage of page loads in Google Chrome according to Chrome Platform Status will be shown instead. Even if some figures are so small that the statistics are not necessarily meaningful, in many cases trends can still be read from the data. Also, these stats can be used as a baseline for future annual editions of this chapter, looking back to see how much the APIs have matured and improved their adoption. Unless otherwise noted, the APIs are only available in Chromium-based browsers, and their specifications are in the early stages of standardization.

Async Clipboard API

With the help of the `document.execCommand()` method, websites could already access the user's clipboard. However, this approach is somewhat restricted, as the API is synchronous (making it difficult to process clipboard items), and it can only interact with selected text in the DOM. This is where the Async Clipboard API (W3C Working Draft) comes in. This new API is not only asynchronous, meaning it doesn't block the page for large chunks of data or waiting for a permission to be granted, but it also allows for images to be copied to or pasted from the clipboard in supported browsers such as Chrome, Edge, and Safari.

Read Access

The Async Clipboard API provides two methods for reading content from the clipboard: a shorthand method for plain text, called `navigator.clipboard.readText()`, and a method for arbitrary data, called `navigator.clipboard.read()`. Currently, most browsers only support HTML content and PNG images as additional data formats. As the clipboard may contain sensitive data, reading from it requires the user's consent.

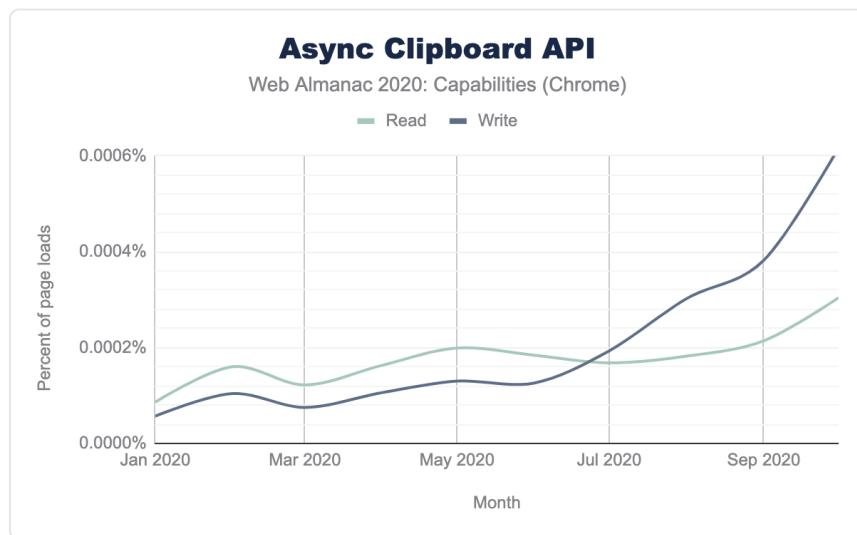


Figure 13.1. Percentage of page loads in Chrome using Async Clipboard API.
(Sources: Async Clipboard Read, Async Clipboard Write)

The Async Clipboard API is comparatively new, so its usage is currently rather low. In March 2020, Safari added support for the Async Clipboard API in Safari 13.1. Over the course of 2020, the usage of the `read()` API was growing. In October 2020, the API was called during

0.0003% of all page loads in Google Chrome.

Write Access

Apart from reading operations, the Async Clipboard API also offers two methods for writing content to the clipboard. Again, there's a shorthand method for plain text, called `navigator.clipboard.writeText()`, and one for arbitrary data called `navigator.clipboard.write()`. In Chromium-based browsers, writing to the clipboard while the tab is active does not require permission. Trying to write to the clipboard when the website is in the background does, however. As this method requires a user gesture and permission first, it's not covered by the HTTP Archive metrics. In contrast to the `read()` method, the `write()` method shows an exponential growth in usage, being part of 0.0006% of all page loads in October 2020.

The Raw Clipboard Access API, another Fugu capability, might even further enhance the Async Clipboard API by allowing arbitrary data to be copied from or pasted to the clipboard.

StorageManager API

Web browsers allow users to store data on the user's system in different ways, such as Cookies, the Indexed Database (IndexedDB), the Service Worker's Cache Storage, or Web Storage (LocalStorage, Session Storage). In modern browsers, developers can easily store hundreds of megabytes and even more, depending on the browser. When browsers run out of space, they can clear data until the system is no longer over the limit, which can lead to data loss.

Thanks to the StorageManager API, which is part of the WHATWG Storage Living Standard, browsers no longer behave like a black box in that regard. This API allows developers to estimate the remaining space available and opt-in to persistent storage, meaning that the browser will not clear a website's data when disk space is low. Therefore, the API introduces a new `StorageManager` interface on the `navigator` object, currently available on Chrome, Edge, and Firefox.

Estimate the available storage

Developers can estimate the available storage by calling `navigator.storage.estimate()`. This method returns a promise resolving to an object with two properties: `usage` shows the number of bytes used by the application and `quota` contains the maximum number of bytes available.

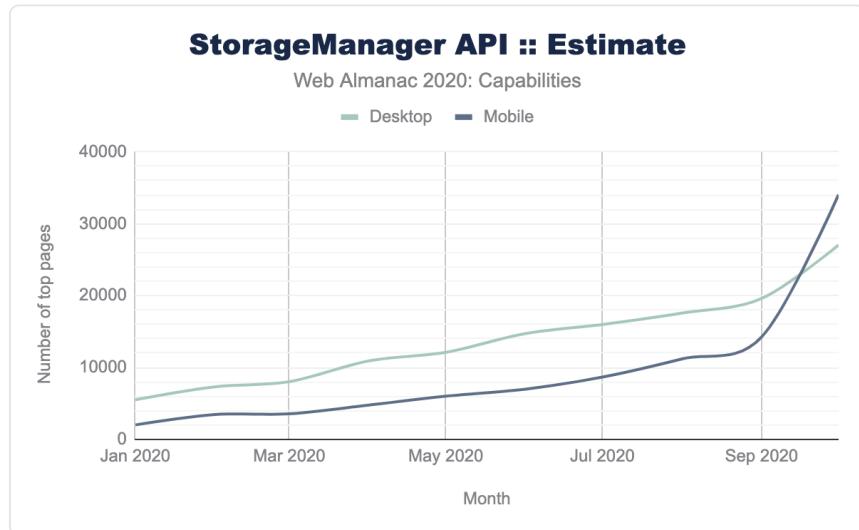


Figure 13.2. Number of pages using the estimate method of the StorageManager API.

The Storage Manager API is supported in Chrome since 2016, Firefox since 2017, and the new Chromium-based Edge. HTTP Archive data shows that the API is used on 27,056 desktop pages (0.49%) and 34,042 mobile pages (0.48%). Over the course of 2020, the usage of the Storage Manager API kept growing. This also makes this interface the most commonly used API in this chapter.

Opt-in to persistent storage

There are two categories of web storage: "Best Effort" and "Persistent", with the first being the default. When a device is low on storage, the browser automatically tries to free up best effort storage. For instance, Firefox and Chromium-based browsers evict storage from the least recently used origins. This, however, poses a risk of losing critical data. To prevent eviction, developers can opt for persistent storage. In this case, the browser will not clear the storage, even when space is low. Users can still choose to clear up the storage manually. To opt for persistent storage, developers need to call the `navigator.storage.persist()` method. Depending on the browser and site engagement, a permission prompt may show, or the request will automatically be accepted or denied.

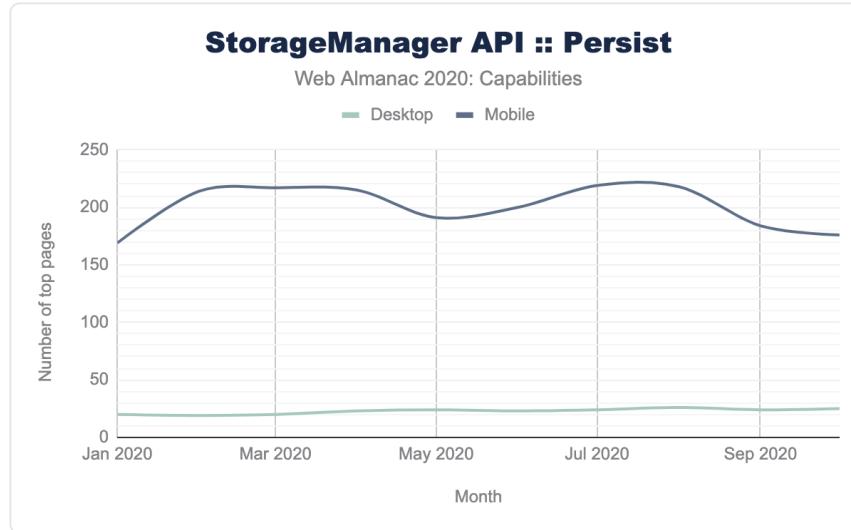


Figure 13.3. Number of pages using the `persist` method of the `StorageManager API`.

The `persist()` API is less often called than the `estimate()` method. Only 176 mobile pages make use of this API, compared to 25 desktop websites. While usage on the desktop seems to remain at this low level, there is no clear trend on mobile devices.

New Notification APIs

With the help of the Push and Notifications APIs, web applications have long been able to receive push messages and display notification banners. However, some parts were missing. Until now, push messages had to be sent via the server; they could not be scheduled offline. Also, web applications installed to the system could not show badges on their icon. The Badging and Notification Triggers APIs enable both scenarios.

Badging API

On several platforms, it's common for applications to present a badge on the application's icon indicating the amount of open actions. For instance, the badge could show the number of unread emails, notifications, or to-do items to complete. The Badging API (W3C Unofficial Draft) allows installed web applications to show such a badge on its icon. By calling `navigator.setAppBadge()`, developers can set the badge. This method takes a number to be shown on the application's badge. The browser then takes care of displaying the closest possible representation on the user's device. If no number is specified, a generic badge will be

shown (e.g., a white dot on macOS). Calling `navigator.clearAppBadge()` removes the badge again. The Badging API is a great choice for email clients, social media apps, or messengers. The Twitter PWA makes use of the Badging API to show the number of unread notifications on the application's badge.

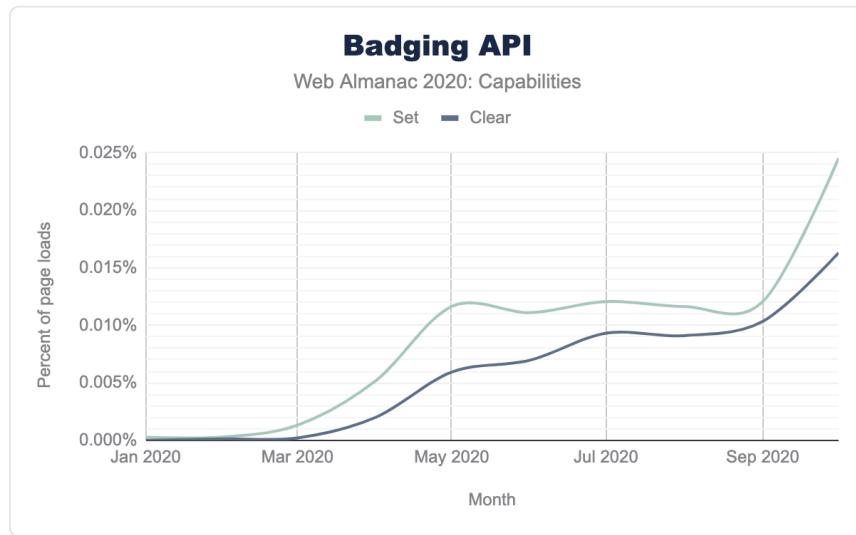


Figure 13.4. Percentage of page loads in Chrome using Badging API.
(Sources: Badge Set, Badge Clear)

In April 2020, Google Chrome 81 shipped the new Badging API, followed by Microsoft Edge 84 in July. After Chrome shipped the API, the usage numbers shot up. In October 2020, on 0.025% of all page loads in Google Chrome, the `setAppBadge()` method is called. The `clearAppBadge()` method is less often called, during around 0.016% of page loads.

Notification Triggers API

The Push API requires the user to be online to receive a notification. Some applications, such as games, reminder or to-do apps, calendars, or alarm clocks, could also determine the target date for a notification locally and schedule it. To support this feature, the Chrome team is experimenting with a new API called Notification Triggers (Explainer, not on a standards track yet). This API adds a new property called `showTrigger` to the `options` map that can be passed to the `showNotification()` method on the Service Worker's registration. The API is designed to allow for different kinds of triggers in the future, albeit for now, only time-based triggers are implemented. For scheduling a notification based on a certain date and time, developers can create a new instance of a `TimestampTrigger` and pass the target timestamp

to it:

```
registration.showNotification('Title', {
  body: 'Message',
  showTrigger: new TimestampTrigger(timestamp),
});
```

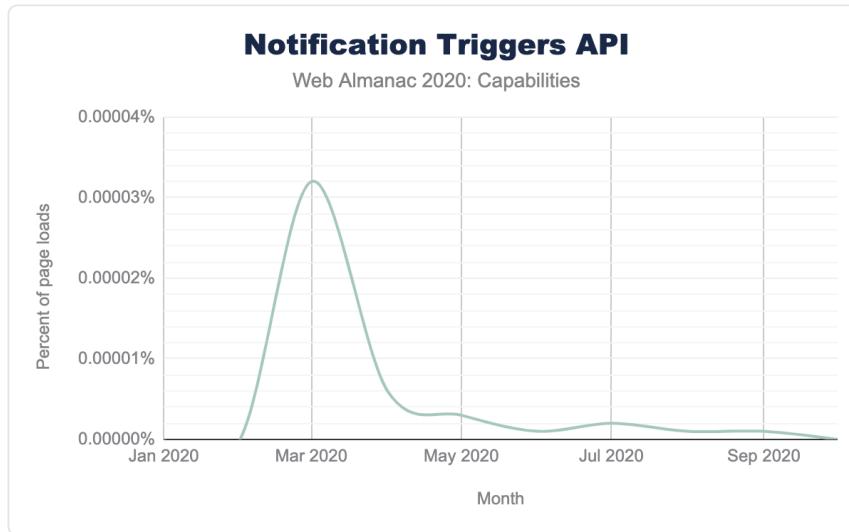


Figure 13.5. Percentage of page loads in Chrome using Notification Triggers API.
(Source: Notification Triggers)

The Fugu team first experimented with Notification Triggers in an origin trial from Chrome 80 to 83, pausing development afterwards due to the lack of feedback by developers. Starting from Chrome 86 released in October 2020, the API has entered the origin trial phase again. This also explains the usage data of Notification Triggers API that peaked at being called on 0.000032% of page loads in Chrome during the first origin trial at around March 2020.

Screen Wake Lock API

To save energy, mobile devices darken the screen backlight and eventually turn off the device's display, which makes sense in most cases. However, there are scenarios where the user may want the application to explicitly keep the display awake, for instance, when reading a recipe while cooking or watching a presentation. The Screen Wake Lock API (W3C Working Draft)

solves this problem by providing a mechanism to keep the screen on.

The `navigator.wakeLock.request()` method creates a wake lock. This method takes a `WakeLockType` parameter. In the future, the Wake Lock API could provide other lock types, such as turning the screen off, but keeping the CPU on. For now, the API only supports screen locks, so there is just one optional argument with the default value of `'screen'`. The method returns a promise that resolves to a `WakeLockSentinel` object. Developers need to store this reference to call its `release()` method and release the screen wake lock later on. The browser will automatically release the lock when the tab is inactive, or the user minimizes the window. Also, the browser may deny a request and reject the promise, for example due to low battery.

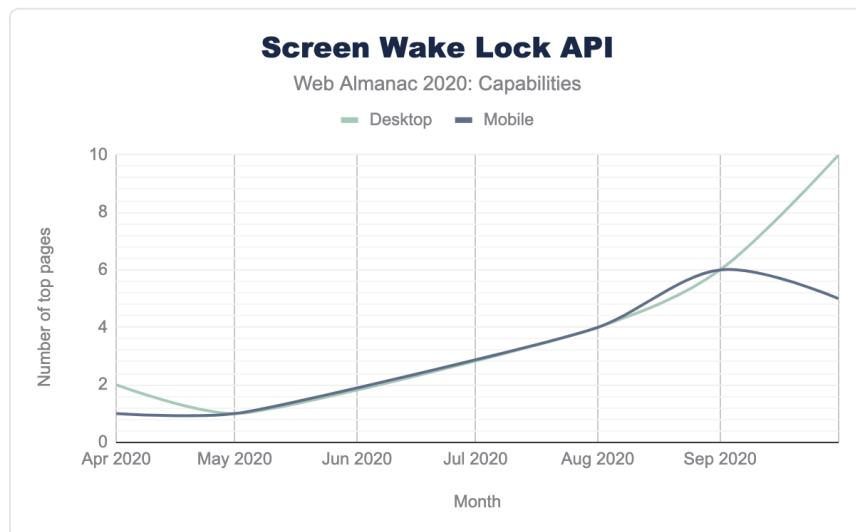


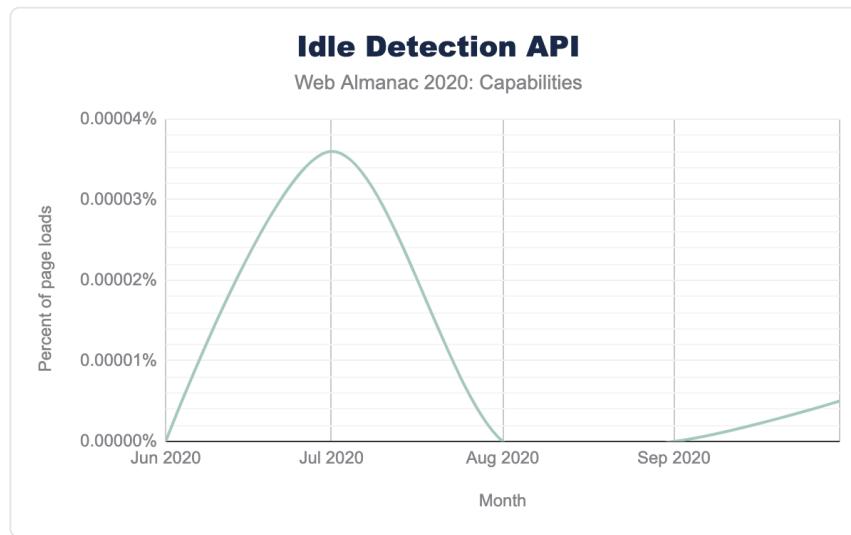
Figure 13.6. Numbers of pages using Screen Wake Lock API.

BettyCrocker.com, a popular cooking website in the US, offers their users an option to prevent the screen from going dark while cooking with the help of the Screen Wake Lock API. In a case study, they published that the average session duration was 3.1 times longer than normal, the bounce rate reduced by 50%, and purchase intent indicators increased by about 300%. The interface therefore has a directly measurable effect on the success of the website or application, respectively. The Screen Wake Lock API shipped with Google Chrome 84 in July 2020. The HTTP Archive only has data for April, May, August, September and October. After the release of Chrome 84, usage rose quickly. In October 2020, the API was adopted on 10 desktop and 5 mobile pages.

Idle Detection API

Some applications need to determine if the user is actively using a device or if they are idle. For instance, chat applications may display that the user is absent. There are various factors that can be taken into account, such as a lack of interaction with the screen, mouse, or keyboard. The Idle Detection API (WICG Draft Community Group Report) provides an abstract API that allows developers to check if either the user is idle or the screen locked, given a certain threshold.

To do so, the API provides a new `IdleDetector` interface on the global `window` object. Before developers can use this functionality, they have to request permission by calling `IdleDetector.requestPermission()` first. If the user grants the permission, developers can create a new instance of `IdleDetector`. This object provides two properties: `userState` and `screenState`, containing the respective states. It will raise a `change` event when either the user's or the screen's state change. Finally, the idle detector needs to be started by calling its `start()` method. The method takes a configuration object with two parameters: a `threshold` defining the time in milliseconds that the user has to be idle (the minimum is a minute), and developers can optionally pass an `AbortSignal` to the `abort` property, which serves to abort idle detection later on.



*Figure 13.7. Percentage of page loads in Chrome using Idle Detection API.
(Source: Idle Detection)*

At the time of this writing, the Idle Detection API is in an origin trial, so its API shape may change in the future. For the same reason, its usage is very low and hardly measurable.

Periodic Background Sync API

When the user closes a web application, it cannot communicate with its backend service anymore. In some cases, developers might still want to synchronize data on a more or less regular basis, just as native applications can. For instance, news applications might want to download the latest headlines before the user wakes up. The Periodic Background Sync API (WICG Draft Community Group Report) strives to bridge this gap between web and native.

Register for periodic sync

The Periodic Background Sync API relies on Service Workers that can run even when the app is closed. As with other capabilities, this API requires users' permission first. The API implements a new interface called `PeriodicSyncManager`. If present, developers can access an instance of this interface on the Service Worker's registration. To synchronize data in the background, the application has to register first, by calling `periodicSync.register()` on the registration. This method takes two parameters: a `tag`, which is an arbitrary string to recognize the registration again later on, and a configuration object that takes a `minInterval` property. This property defines the desired minimum interval in milliseconds by the developer. However, the browser ultimately decides how often it will actually invoke background synchronization:

```
registration.periodicSync.register('articles', {
  minInterval: 24 * 60 * 60 * 1000 // one day
});
```

Respond to a periodic sync interval

For each tick of the interval, and if the device is online, the browser triggers the Service Worker's `periodicsync` event. Then, the Service Worker script can perform the necessary steps to synchronize the data:

```
self.addEventListener('periodicsync', (event) => {
  if (event.tag === 'articles') {
    event.waitUntil(syncStuff());
  }
});
```

At the time of this writing, only Chromium-based browsers implement this API. On these browsers, the application has to be installed first (i.e., added to the homescreen) before the API can be used. The site engagement score of the website defines if and how often periodic sync events can be invoked. In the current conservative implementation, websites can sync content once a day.

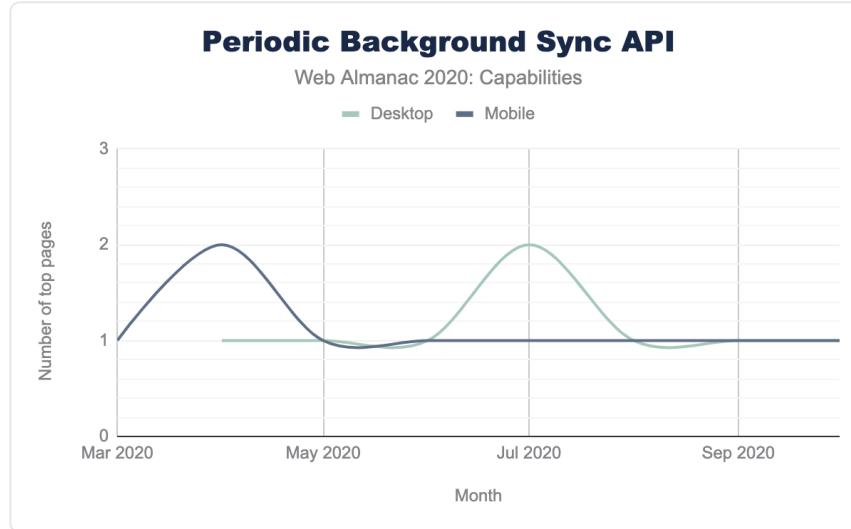


Figure 13.8. Number of pages using Periodic Background Sync API.

The use of the interface is currently very low. Over 2020, only one or two pages monitored by HTTP Archive made use of this API.

Integration with native app stores

PWAs are a versatile application model. However, in some cases, it may still make sense to offer a separate native application: for example, if the app needs to use features that are not available on the web, or based on the programming experience of the app developer team. When the user already has a native app installed, apps might not want to send notifications twice or promote the installation of a corresponding PWA.

To detect if the user already has a related native application or PWA on the system, developers can use the `getInstalledRelatedApps()` method (WICG Draft Community Group Report) on the `navigator` object. This method is currently provided by Chromium-based browsers and works for both Android and Universal Windows Platform (UWP) apps. Developers need to adjust the native app bundles to refer to the website and add information about the native

app(s) to the Web App Manifest of the PWA. Calling the `getInstalledRelatedApps()` method will then return the list of apps installed on the user's device:

```
const relatedApps = await navigator.getInstalledRelatedApps();
relatedApps.forEach((app) => {
  console.log(app.id, app.platform, app.url);
});
```

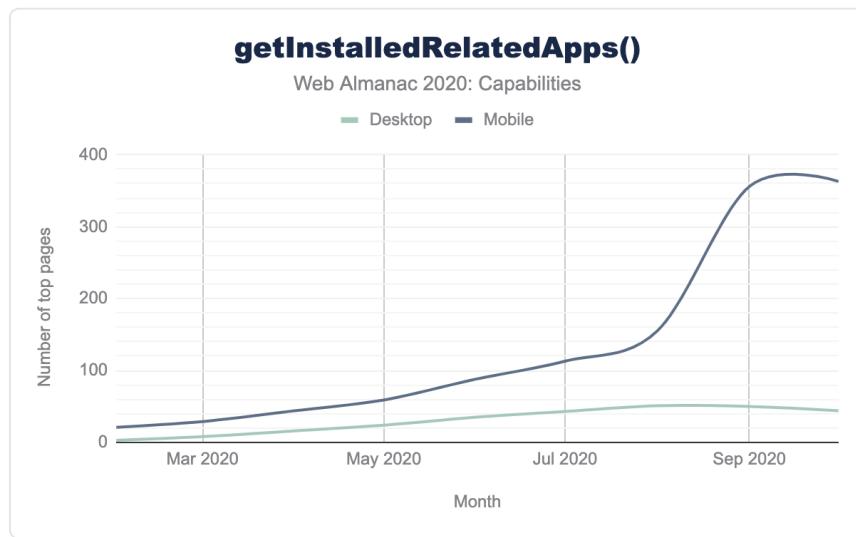


Figure 13.9. Number of pages using `getInstalledRelatedApps()`.

Over the course of 2020, the `getInstalledRelatedApps()` API shows a steady growth on mobile websites. In October, 363 mobile pages tracked by the HTTP Archive made use of this API. On desktop pages, the API does not grow quite as fast. This could also be due to Android stores currently providing significantly more apps than the Microsoft Store does for Windows.

Content Indexing API

Web apps can store content offline using various ways, such as Cache Storage, or IndexedDB. However, for users it's hard to discover which content is available offline. The Content Indexing API (WICG Editor's Draft) allows developers to expose content more prominently. Currently, Chrome on Android is the only browser to support this API. This browser shows a list of "Articles for you" in the Downloads menu. Content indexed via the Content Indexing API will

appear there.

The Content Indexing API extends the Service Worker API by providing a new `ContentIndex` interface. This interface is available via the `index` property of the Service Worker's registration. The `add()` method allows developers to add content to the index: Each piece of content must have an ID, a URL, a launch URL, title, description, and a set of icons. Optionally, the content can be grouped into different categories such as articles, homepages, or videos. The `delete()` method allows for removing content from the index again, and the `getAll()` method returns a list of all indexed entries.

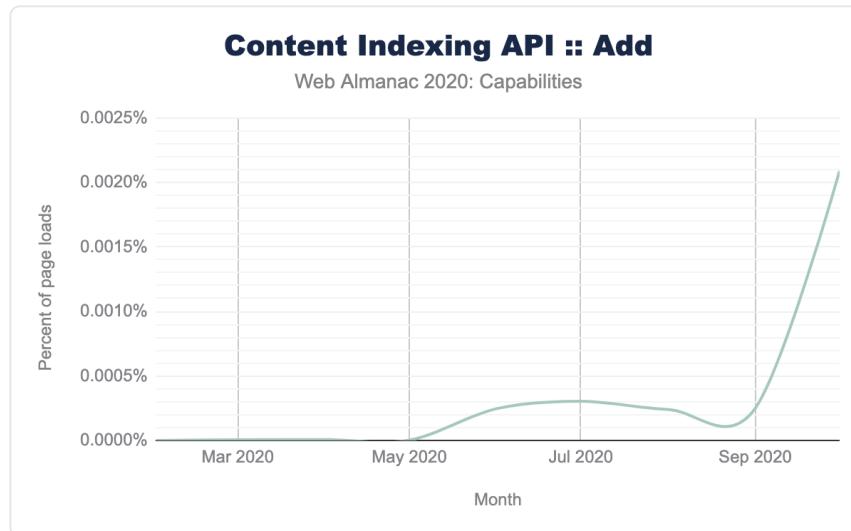


Figure 13.10. Percentage of page loads in Chrome using Content Indexing API.
(Source: Content Indexing)

The Content Indexing API launched with Chrome 84 in July 2020. Directly after shipping, the API was used during approximately 0.0002% of page loads in Chrome. In October 2020, this value has increased almost tenfold.

New Transport APIs

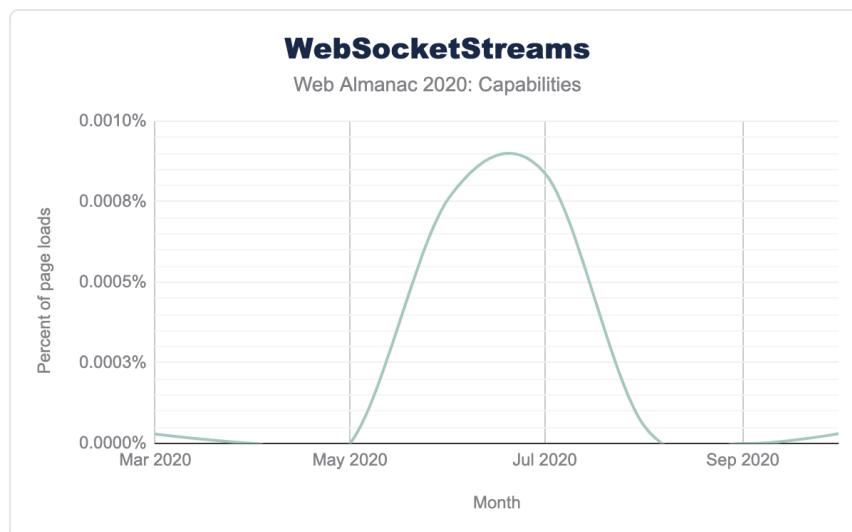
Finally, there are two new transport methods that are currently in origin trial. The first one allows developers to receive high-frequency messages with WebSockets, while the second one introduces an entirely new bidirectional communication protocol apart from HTTP and WebSockets.

Backpressure for WebSockets

The WebSocket API is a great choice for bidirectional communication between websites and servers. However, the WebSocket API does not allow for backpressure, so applications dealing with high-frequency messages may freeze. The `WebSocketStream` API (Explainer, not on the standards track yet) wants to bring easy-to-use backpressure support to the WebSocket API by extending it with streams. Instead of using the usual `WebSocket` constructor, developers need to create a new instance of the `WebSocketStream` interface. The `connection` property of the stream returns a promise that resolves to a readable and writable stream that allow to obtain a stream reader or writer, respectively:

```
const wss = new WebSocketStream(WSS_URL);
const {readable, writable} = await wss.connection;
const reader = readable.getReader();
const writer = writable.getWriter();
```

The `WebSocketStream` API transparently solves backpressure, as the stream readers and writers will only proceed if it's safe to do so.



*Figure 13.11. Percentage of page loads in Chrome using `WebSocketStreams`.
(Source: `WebSocketStream`)*

The `WebSocketStream` API has completed its first origin trial and is now back in the

experimentation phase again. This also explains why the usage of this API currently is so low that it's hardly measurable.

Make it QUIC

QUIC (IETF Internet-Draft) is a multiplexed, stream-based, bidirectional transport protocol implemented on UDP. It's an alternative to HTTP/WebSocket APIs that are implemented on top of TCP. The QuicTransport API is the client-side API for sending messages to and receiving messages from a QUIC server. Developers can choose to send data unreliably via datagrams, or reliably by using its streams API:

```
const transport = new QuicTransport(QUIC_URL);
await transport.ready;

const ws = transport.sendDatagrams();
const stream = await transport.createSendStream();
```

QuicTransport is a valid alternative to WebSockets, as it supports the use cases from the WebSocket API and adds support for scenarios where minimal latency is more important than reliability and message order. This makes it a good choice for games and applications dealing with high-frequency events.

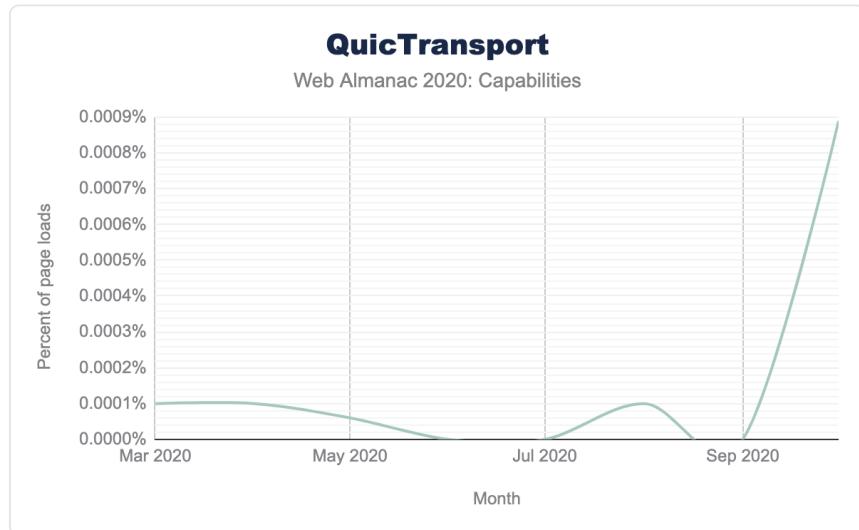


Figure 13.12. Percentage of page loads in Chrome using QuicTransport.
(Source: QuicTransport)

The use of the interface is currently still so low that it's hardly measurable. In October 2020, it has increased strongly and is now used during 0.00089% of page loads in Chrome.

Conclusion

The state of web capabilities in 2020 is healthy, as new, powerful APIs regularly ship with new releases of Chromium-based browsers. Some interfaces like the Content Indexing API or Idle Detection API help to add finishing touches to certain web applications. Other APIs, such as the File System Access and Async Clipboard API, allow a whole new application category, namely productivity apps, to finally fully make the shift to the web. Some APIs such as Async Clipboard and Web Share API have already made their way into other, non-Chromium browsers. Safari even was the first mobile browser to implement the Web Share API.

Through its rigorous process, the Fugu team ensures that access to these features takes place in a secure and privacy-friendly manner. Additionally, the Fugu team actively solicits the feedback from other browser vendors and web developers. While the usage of most of these new APIs is comparatively low, some APIs presented in this chapter show an exponential or even hockey stick-like growth, such as the Badging or Content Indexing API. The state of web capabilities in 2021 will depend on the web developers themselves. The author encourages the community to build great web applications, make use of the powerful APIs in a backwards-compatible manner, and help make the web a more capable platform.

Author



Christian Liebel

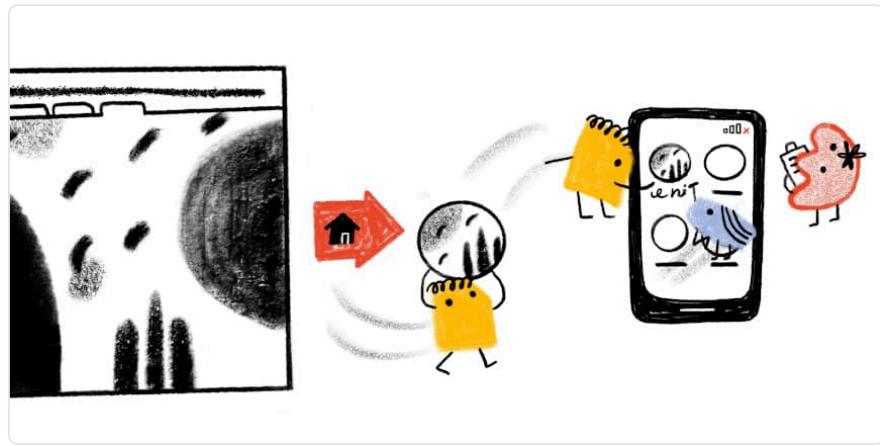
🐦 @christianliebel Ⓛ christianliebel ⌂ <https://christianliebel.com>

Christian Liebel is a consultant at Thinktecture⁴⁷, supporting clients from various business areas in implementing first-class web applications. He is a Microsoft MVP for Developer Technologies, Google GDE for Web/Capabilities and Angular, and participates in the W3C Web Applications Working Group.

47. <https://thinktecture.com>

Part II Chapter 14

PWA [UNEDITED]



Written by [hemanth.hm](#)

Reviewed by [Pascal Schilp](#), [Jad Joubran](#), [Pearl Latteier](#), and [Gokulakrishnan Kalaikovan](#)

Analyzed by [Barry Pollard](#)

Introduction

In 1990 we had the first ever browser called the “WorldWideWeb” and ever since the web and the browser have been evolving and for the web to progress itself into a native behavior is a big win especially in this era of mobile domination. URLs and web browsers have provided a ubiquitous way to distribute information and so a technology which provides native app capabilities to the browser is a game changer. Progressive Web Apps provide such advantages for the web to compete with other applications.

Simply put, a web application which give native-like application experience can be considered as a PWA,. It is built using common web technologies including HTML, CSS and JavaScript and can operate seamlessly across devices and environments on a standards-compliant browser.

The crux of a progressive web app is the *service worker*, which can be thought of as a proxy sitting between the browser and user. A service worker gives the developer total control over the network, rather than the network controlling the application.

As last year's chapter stated, it started in December 2014 when Chrome 40 first implemented the meat of what is now known as Progressive Web Apps (PWA). This was a collaborative effort for the web standards body and the term PWA was coined by Frances Berriman and Alex Russell in 2015.

In this chapter of Web Almanac we will be looking into each of the components that make PWA what it is, from a data-driven perspective.

Service workers

Service workers are at the very center of the progressive web apps and can be thought as a proxy servers between web applications running in browsers and the network. They help developers control the network requests rather than the network requests controlling the application.

Service worker usage

From the data we gathered it was derived that about **0.88%** desktop sites and **0.87%** mobile sites use a service worker. This was for the month of August 2020 and, to put that into perspective, that equates to 49,305 (out of 5,593,642) desktop sites and 55,019 (out of 6,347,919) mobile sites. While that usage may seem low, it is important that we realize that other measurements equate that to **16.6%** of the web traffic—the difference being due to high traffic websites tending to use service workers more.

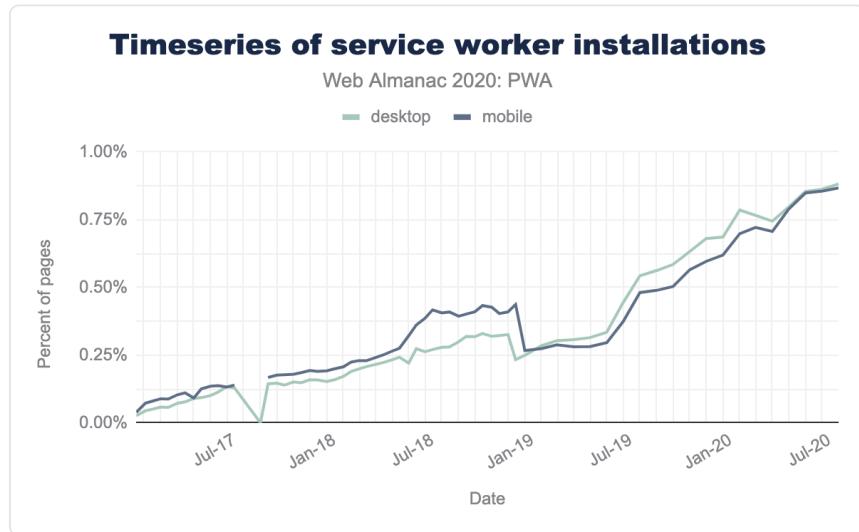


Figure 14.1. Timeseries of Service Worker installation.

Lighthouse insights

Lighthouse provides automated auditing, performance metrics, and best practices for the web and has been instrumental in shaping web's performance. We looked at the PWA category audits gathered for over 6,811,475 pages and this has given us great insights on a few important touch points.

Lighthouse Audit	Weight	Percentage
<i>load-fast-enough-for-pwa</i>	7	27.97%*
<i>works-offline</i>	5	0.86%
<i>installable-manifest</i>	2	2.21%
<i>is-on-https</i>	2	66.67%
<i>redirects-http</i>	2	70.33%
<i>viewport</i>	2	88.43%
<i>apple-touch-icon</i>	1	34.75%
<i>content-width</i>	1	79.37%
<i>maskable-icon</i>	1	0.11%
<i>offline-start-url</i>	1	0.75%
<i>service-worker</i>	1	1.03%
<i>splash-screen</i>	1	1.90%
<i>themed-omnibox</i>	1	4.00%
<i>without-javascript</i>	1	97.57%

Figure 14.2. Lighthouse PWA audits.

Note the Lighthouse performance stats were incorrect for our August crawl so the *load-fast-enough-for-pwa* stat has been replaced with September results.

A fast page load ensures a good mobile user experience, particularly when slower cellular network's are taken into consideration.

27.56% of pages loaded fast enough for a PWA. Given how geographically distributed the web is, having a fast load time with lighter pages matter the most of the next billion users of the web, most of whom will be introduced to the internet via a mobile device.

If you're building a Progressive Web App, consider using a service worker so that your app can work offline **0.92%** of pages were offline ready.

Browsers can proactively prompt users to add your app to their homescreen, which can lead to higher engagement. **2.21%** of pages had an Installable manifest. Manifest plays an important role in how the application starts, the looks and feel of the icon on the homescreen and as an

impact on the engagement rate directly.

All sites should be protected with HTTPS, even ones that don't handle sensitive data. This includes avoiding mixed content, where some resources are loaded over HTTP despite the initial request being served over HTTPS. HTTPS prevents intruders from tampering with or passively listening in on the communications between your app and your users, and is a prerequisite for HTTP/2 and many new web platform APIs. Learn more about [is-on-https](#) check. **67.27%** of sites were on HTTPS and it is surprising that we haven't reached there yet. This number is pretty decent and will get better as browsers mandate the applications to be on HTTPS and scrutinize those which are not on HTTPS.

If you've already set up HTTPS, make sure that you redirect all HTTP traffic to HTTPS in order to enable secure connection the users without changing the URL **69.92%** of the sites redirects HTTP. Redirecting all the HTTP to HTTPS on your application should be simple steps towards robustness, though the HTTP redirection to HTTPS has a decent number, it can do better.

By adding `<meta name="viewport">` tag to optimize your app for mobile screens. **88.43%** of the sites have the viewport meta tag. It is not surprising that the usage of viewport meta tag is on the higher side as most of the applications are aware and getting there in terms of viewport optimization.

For ideal appearance on iOS, your progressive web app should define an `apple-touch-icon` meta tag. It must point to a non-transparent 192px (or 180px) square PNG. **32.34%** of the sites use the apple touch icon.

If the width of your app's content doesn't match the width of the viewport, your app might not be optimized for mobile screens. **79.18%** of the sites have the content-width set.

A maskable icon ensures that the image fills the entire shape without being letterboxed when adding the progressive web app to the home screen. Only **0.11%** of sites use this, but given that it is a brand new feature, having any usage here is encouraging. As it is a new feature we were expecting the numbers to be very low and are expected to improve in the coming years.

A service worker enables your web app to be reliable in unpredictable network conditions. **0.77%** of sites has an offline start URL.

The service worker is the feature that enables your app to use many Progressive Web App features, such as offline usage and push notifications. **1.05%** of pages have service workers enabled. Service worker helps to achieve offline support, which is the most important feature for a PWA, as flaky networks are the most common issue that the users of web applications face. Given that this can be addressed with service workers, it is surprising that number is still so low.

A themed splash screen ensures a native like experience when users launch your app from their homescreens. **1.95%** of pages had splash screens.

The browser address bar can be themed to match your site. **3.98%** of pages had themed omnibox.

Your app should display some content when JavaScript is disabled, even if it's just a warning to the user that JavaScript is required to use the app. **96.23%** pages can work with JavaScript disabled.

Service worker events

In a service worker one can listen for a number of events:

1. `install`, which occurs upon service worker installation.
2. `activate`, which occurs upon service worker activation.
3. `fetch`, which occurs whenever a resource is fetched.
4. `push`, which occurs when a push notification arrives.
5. `notificationclick`, which occurs when a notification is being clicked.
6. `notificationclose`, which occurs when a notification is being closed.
7. `message`, which occurs when a message sent via `postMessage()` arrives.
8. `sync`, which occurs when a background sync event occurs.

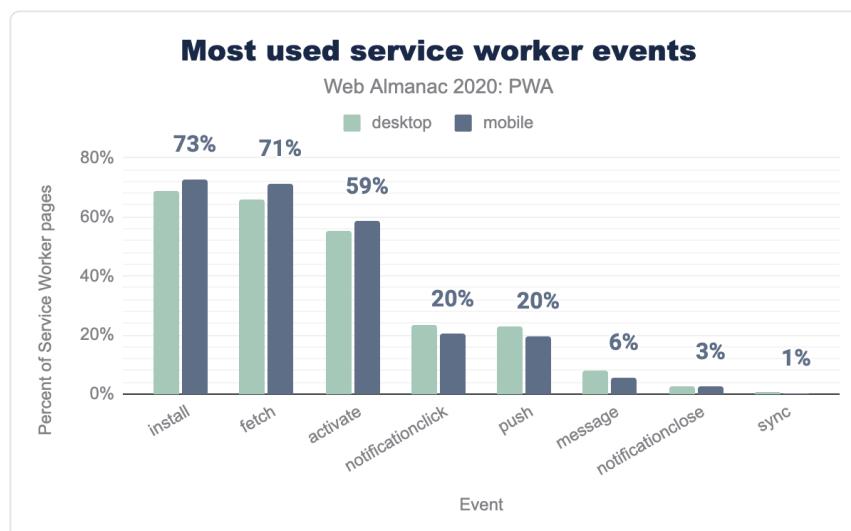


Figure 14.3. Most used service worker events.

We have examined which of these events are being listened to by service workers in our data set. The results for mobile and desktop are very similar with `install`, `fetch`, and `activate` being the three most popular events, followed by `message`, `notification`, `click`, `push` and `sync`. If we interpret these results, offline use cases that service workers enable are the most attractive feature for app developers, far ahead of push notifications. Due to its limited availability, and less common use case, background sync doesn't play a significant role at this time.

Web app manifests

The web app manifest is a JSON-based file that provides developers with a centralized place to put metadata associated with a web application, it dictates how the application should behave on desktop or mobile in terms of the icon, orientation, theme color and likes.

Having a web app manifest does not necessarily indicate the site is a progressive web app, as they can exist independently of service worker usage. However, as we are interesting PWAs in this chapter, we have investigated only those manifests for sites where a service worker also exists. This is different than the approach taken in last year's PWA chapter which looked at overall manifest usage so you may notice some differences in results this year.

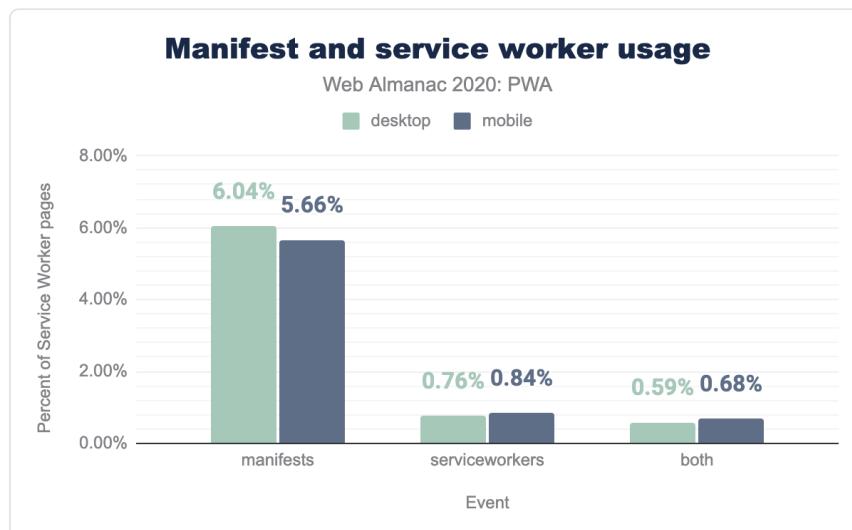


Figure 14.4. Manifest and service worker usage.

Manifest Properties

Web manifest dictates the applications meta properties. We looked at the different properties defined by the Web App Manifest specification, and also considered non-standard proprietary properties. According to the spec, the following properties are valid properties:

1. `background_color`
2. `categories`
3. `description`
4. `dir`
5. `display`
6. `iarc_rating_id`
7. `icons`
8. `lang`
9. `name`
10. `orientation`
11. `prefer_related_applications`
12. `related_applications`
13. `scope`
14. `screenshots`
15. `short_name`
16. `shortcuts`
17. `start_url`
18. `theme_color`

There were very little differences between mobile and desktop stats.

The proprietary properties we encountered frequently were `gcm_sender_id` used by Google Cloud Messaging (GCM) service. We also found other interesting attributes like:

`browser_action`, `DO_NOT_CHANGE_GCM_SENDER_ID` (which was basically a comment, used as JSON doesn't allow comments), `scope`, `public_path`, `cacheDigest`.

On both platforms, however, there's a long tail of properties that are not interpreted by browsers yet contain potentially useful metadata.

We also found a non-trivial amount of mistyped properties; our favorite ones being variation of `theme-color`, `Theme_color`, `theme-color`, `Theme_color` and `orientation`.

In order for a PWA to be fruitful it needs to have a manifest and a service worker. It is interesting to note that manifests are used a lot more than service workers. This is due, in large part, to the fact that CMS like WordPress, Drupal and Joomla have manifests by default.

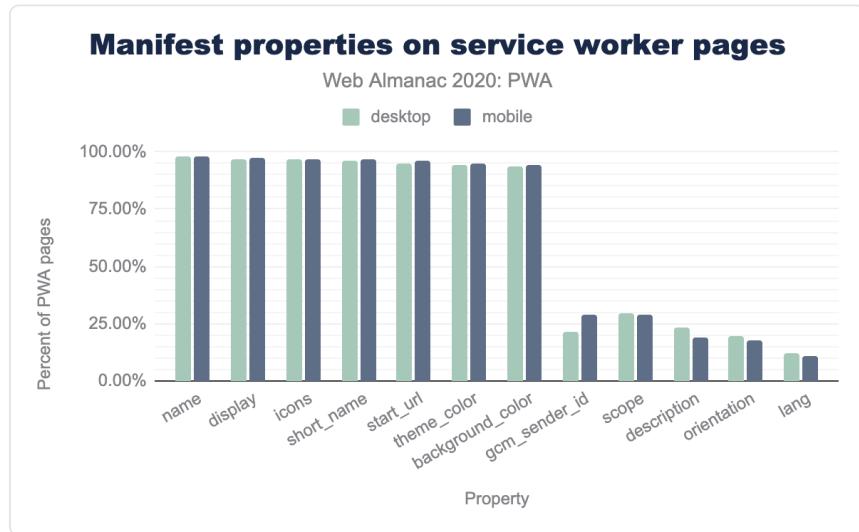


Figure 14.5. Manifest properties on service worker pages.

Top Manifest display values

Out of the five most common `display` values, `standalone` dominated the list with 86.73% of desktop and 89.28% of mobile pages using this. This isn't surprising at all as this mode provides the native app-like feel. Next in the list was `minimal-ui` with 6.30% of desktop and 5.00% of mobile sites opting for them. This is similar to `standalone` except for the fact that some browser UI is retained.

Most used display values for service worker pages

Web Almanac 2020: PWA

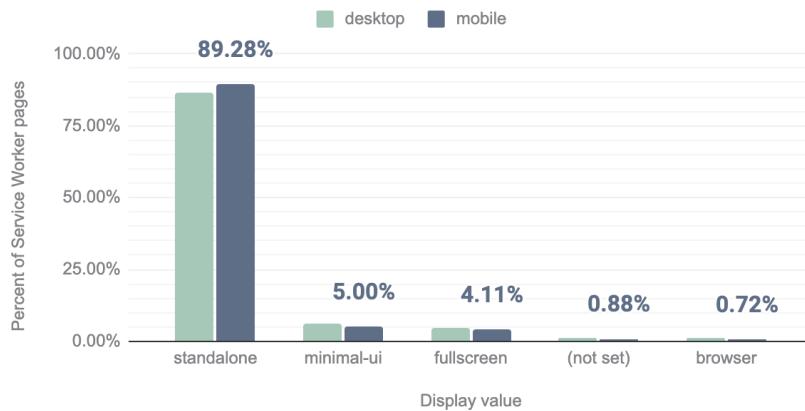


Figure 14.6. Most used `display` values for service worker pages.

Top manifest categories

Out of all the top `categories`, shopping stood at the top at with `13.16%` on the mobile traffic, which is not unexpected as PWAs are e-commerce applications. News was next with `5.26%` on the mobile traffic.

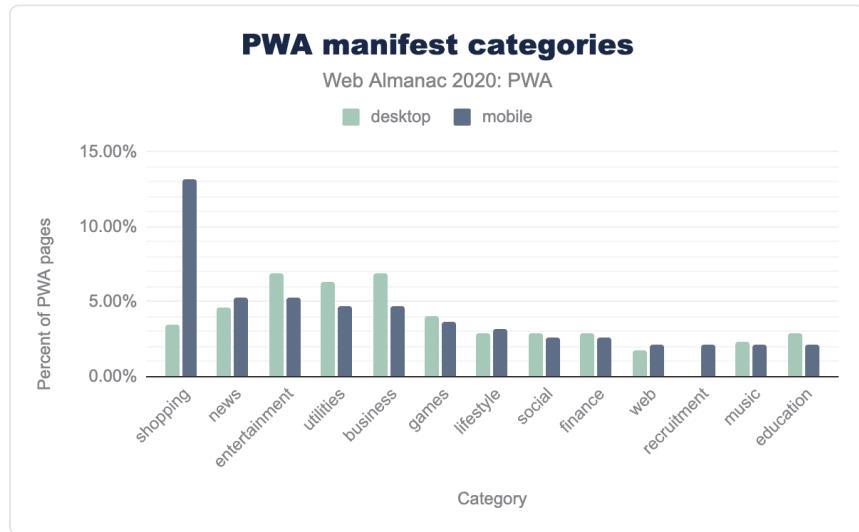


Figure 14.7. PWA manifest categories.

Manifests preferring native

98.24% and 98.52% of mobile sites set the `preferred_related_applications` manifest property to not prefer native apps, but instead use web version where they exist. For the small percentage where this is set to `true` this is a signal that there are many web applications that just have a manifest but aren't really full PWAs yet.

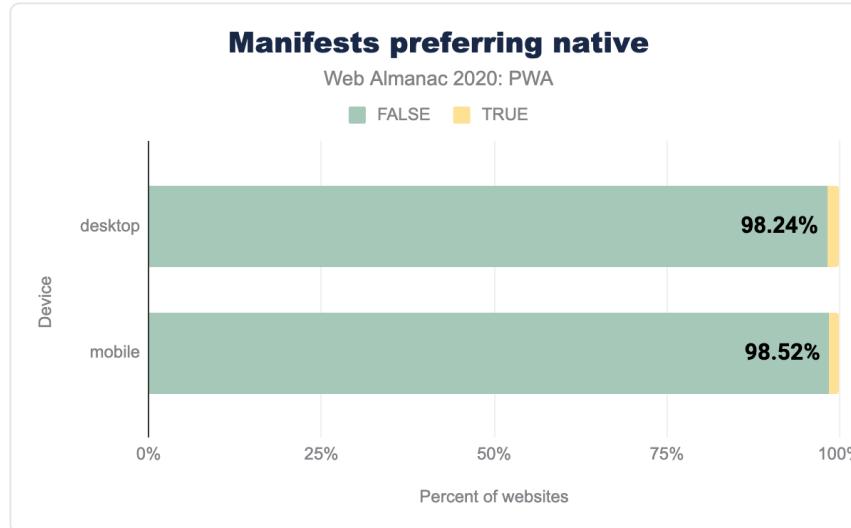


Figure 14.8. Manifest preferring native.

Top manifest icon sizes

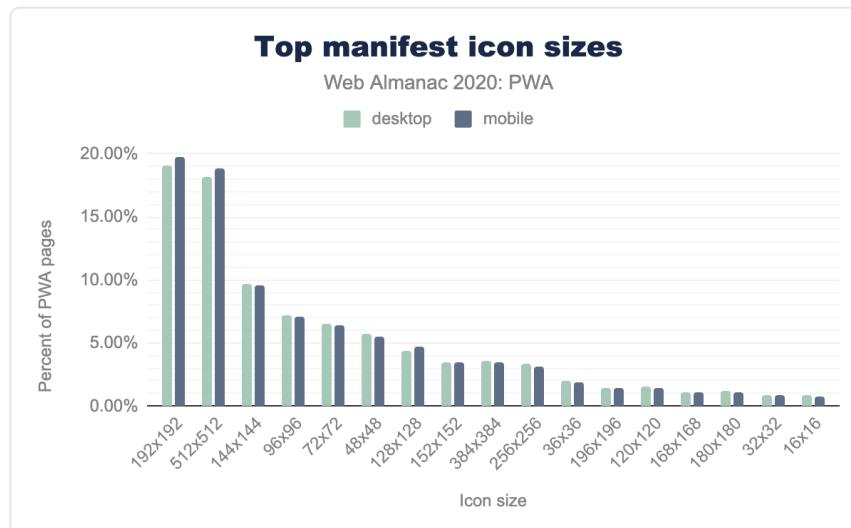


Figure 14.9. Top manifest icon sizes.

Lighthouse requires at least an icon sized 192x192 pixels, but common favicon generation tools

create a plethora of other sizes, too. It is always better to use the recommended icon sizes for each device so it is encouraging to see such a wide spread usage of different icon sizes.

Top manifest orientations

The valid values for the orientation property are defined in the Screen Orientation API specification. Currently, they are:

1. `any`
2. `natural`
3. `landscape`
4. `portrait`
5. `portrait-primary`
6. `portrait-secondary`
7. `landscape-primary`
8. `landscape-secondary`

Out of which we noticed that `portrait`, `any` and `portrait-primary` properties took precedence.

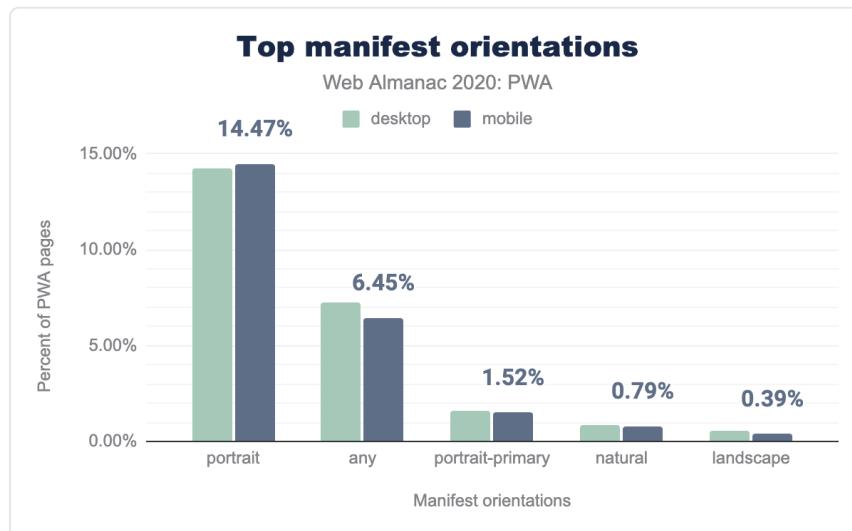


Figure 14.10. Top manifest orientations.

Service worker libraries

There are many cases, where the service workers use libraries as dependencies, be it external dependencies or the application's internal dependencies. These are usually fetched to the service worker via `importScripts()` API, in this section we will look into stats on such libraries.

Popular import scripts

The `importScripts()` API of the `WorkerGlobalScope` interface synchronously imports one or more scripts into the worker's scope, the same is used to import external dependencies to the service worker.

client	desktop	mobile
<i>Uses <code>importScripts()</code></i>	29.60%	23.76%
<i>Workbox</i>	17.70%	15.25%
<i>sw_toolbox</i>	13.92%	12.84%
<i>firebase</i>	3.40%	3.09%
<i>OneSignalSDK</i>	4.23%	2.76%
<i>najva</i>	1.89%	1.52%
<i>upush</i>	1.45%	1.23%
<i>cache_polyfill</i>	0.70%	0.72%
<i>analytics_helper</i>	0.34%	0.39%
<i>Other Library</i>	0.27%	0.15%
<i>No Library</i>	58.81%	64.44%

Figure 14.11. PWA library usage.

Around 30% of the desktop and 25% of mobile sites uses `importScripts()`, of which `workbox`, `sw_toolbox` and `firebase` take the first three positions respectively.

Workbox usage

Out of many libraries available, Workbox was the most heavily used with an adoption rate of **12.86%** and **15.29%** of PWA sites on mobile and desktop respectively.

Out of many methods that Workbox provides, we noticed that **strategies** were used by **29.53%** of desktop and **25.71%** on mobile, **routing** followed it with **18.91%** and **15.61%** adoption and finally **precaching** were next most used with **16.54%** and **12.98%** on desktop and mobile respectively.

This indicated that the strategies API, as one of the most complicated requirements for the developers, played a very important role when they decided to code themselves or rely on libraries like Workbox.

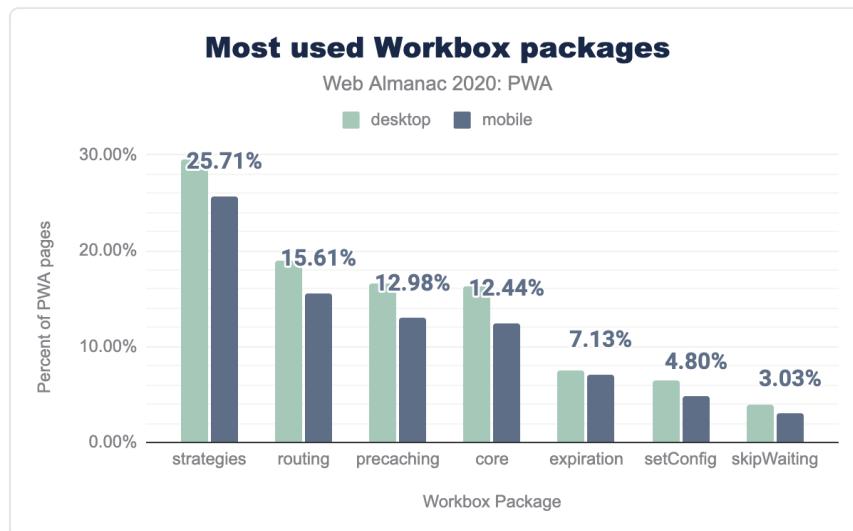


Figure 14.12. Most used Workbox packages.

Conclusion

The stats in this chapter show that PWAs are still continuing to grow in adoption, due to the advantages they give for performance and greater control over caching particularly for mobile. With those advantages and ever increasing capabilities, means we still have a lot of potential for growth and when compared to 2019. We expect to see even more progress in 2021!

More and more browsers and platforms are supporting the technologies powering PWAs. This

year, we saw that Edge gained support for the Web App Manifest. Depending on your use case and target market, you may find that the majority of your users (close to 96%) have PWA support. That is a great improvement! In all cases, it's important to approach technologies such as Service Worker, Web App Manifest should be treated as progressive enhancement. Where you can provide an exceptional user experience through these technologies whenever possible. With the above stats, we're excited for another year of PWA growth!

Author



hemanth.hm

 @gnumanth  hemanth  <https://h3manth.com>

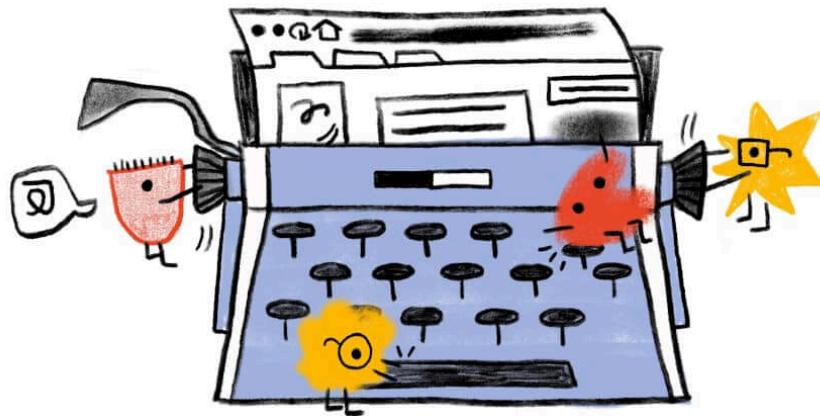
Hemanth HN⁴⁸ is a FOSS Computer polyglot, FOSS philosopher, GDE for web and payments domain, DuckDuckGo community member, TC39 delegate and Google Launchpad Accelerator mentor. Loves The WEB && CLI. Hosts TC39er.us⁴⁹ podcast.

48. <https://h3manth.com>

49. <https://TC39er.us>

Part III Chapter 15

CMS



Written by Alex Denning

Reviewed by Jonathan Wold, Renee Johnson, and Alberto Medina

Analyzed by Greg Brimble and Rick Viscomi

Introduction

The term Content Management System (CMS) refers to systems enabling individuals and organizations to create, manage, and publish content. A CMS for web content, specifically, is a system aimed at creating, managing, and publishing content to be consumed and experienced via the internet.

Each CMS implements some subset of a wide range of content management capabilities and the corresponding mechanisms for users to build websites easily and effectively around their content. Content is often stored in a type of database, providing users with the flexibility to reuse it wherever needed for their content strategy. CMSs also provide administrative capabilities aimed at making it easy for users to upload and manage content as needed.

There is great variability on the type and scope of the support CMSs provide for building sites; some provide ready-to-use templates which are supplemented with user content, and others require much more user involvement for designing and constructing the site structure.

When we think about CMSs, we need to account for all the components that play a role in the viability of such a system for providing a platform for publishing content on the web. All of these components form an ecosystem surrounding the CMS platform, and they include hosting providers, extension developers, development agencies, site builders, etc. Thus, when we talk about a CMS, we usually refer to both the platform itself and its surrounding ecosystem.

There are many interesting and important aspects to analyze and questions to answer in our quest to understand the CMS space and its role in the present and the future of the web. We acknowledge the vastness and complexity of the CMS platforms space and bring to it our curiosity along with deep expertise on some of the major players in the space.

In this chapter, we seek to help understand the current state of the CMS ecosystems, the role they play in shaping users' perception of how content can be consumed and experienced on the web, and their impact on the environment. Our goal is to discuss aspects related to the CMS landscape in general, and the characteristics of web pages generated by these systems.

This second edition of the Web Almanac builds on last year's work. We now have the benefit of being able to compare the 2020 results to 2019 in order to start establishing trends. Let's dive into our analysis.

Why use a CMS in 2020?

People and organizations use a CMS in 2020 as in many cases CMSs offer a shortcut to creating a website which meets their needs. As we'll discuss later, there are both general and specialized CMSs. The general CMSs are often extensible through add-ons, and the specialized CMSs are often focused on specific industry needs or functionality.

Whichever CMS used, it is in use because it solves a problem for the user or organization. It's beyond our scope to explore why each CMS is chosen, but later we do explore why the most popular CMS, WordPress, is disproportionately chosen.

CMS adoption

Our analysis throughout this work looks at desktop and mobile websites. The vast majority of URLs we looked at are in both datasets, but some URLs are only accessed by desktop or mobile devices. This can cause small divergences in the data, and we thus look at desktop and mobile results separately.

More than 42% of web pages are powered by a CMS platform, an increase of over 5% from 2019. This breaks down to 42.18% on desktop, up from 40.01% in 2019, and 42.27% on mobile,

up from 39.61% in 2019.

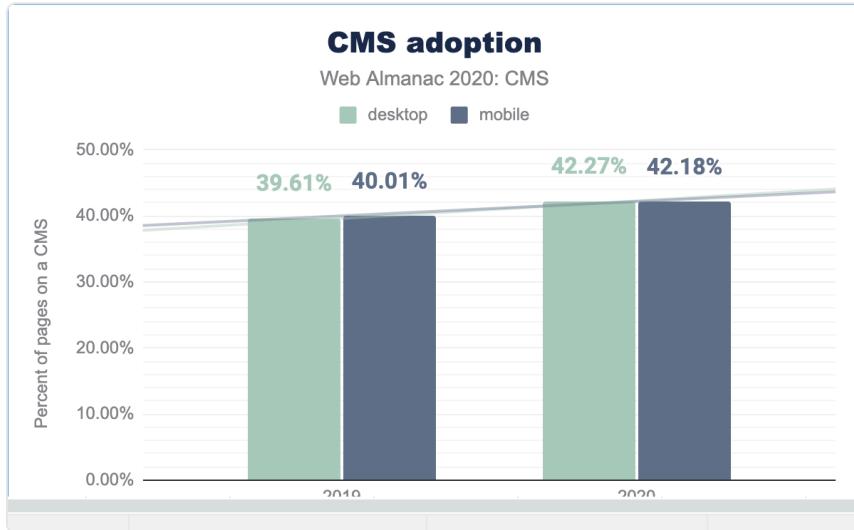


Figure 15.1. CMS adoption trend.

Year	Desktop	Mobile
2019	39.61%	40.01%
2020	42.27%	42.18%
% Change	6.71%	5.43%

Figure 15.2. CMS adoption statistics.

The increase in desktop web pages powered by a CMS platform is 5.43% from last year. On mobile this increase is roughly a quarter higher, at 6.71%.

As with last year, we see different results from other datasets for tracking market share of CMS platforms, such as W3Techs. W3Techs reports at the time of writing that 60.6% of web pages are created by CMSs, up from 56.4% a year ago. This is a 6.4% increase, which broadly matches our findings.

The deviation between our analysis and W3Techs' analysis can be explained by a difference in research methodologies. You can read more about ours on the [Methodology page](#).

Our research identified 222 individual CMSs, with these ranging from a single install to millions on a single CMS.

Some of them are open source (e.g. WordPress, Joomla, others) and some of them are proprietary (e.g. Wix, Squarespace, others). As we'll discuss later, the top 3 CMSs by adoption share are all open source, but proprietary platforms have seen large increases in adoption share this year. Some CMS platforms can be used on "free" hosted or self-hosted plans, and there are also options for using these platforms on higher-tiered plans even at the enterprise level.

The CMS space as a whole is a complex, federated universe of CMS ecosystems, all separated and at the same time intertwined. Our research shows CMSs are only getting more important. The minimum of 5% increase in adoption of CMSs shows that in a year when COVID-19 has created immense uncertainty, solid CMS platforms have provided some stability. As we discussed last year, these platforms play a key role for us to succeed in our collective quest for an evergreen, healthy, and vibrant web. This has become truer since, and we expect it to continue to be the case going forward.

Top CMSs

Our analysis counted 222 separate CMSs. While this is a high count, 204 (92%) of these have an adoption share of 0.01% or lower. This leaves only 13 CMSs with an adoption share of between 0.1 and 1%, and four with a share of between 1 and 2%, and one with a share over this.

The one CMS with a share over 2% is WordPress, which has a 31% usage share. This is over 15 times the share of the next most popular CMS, Joomla:

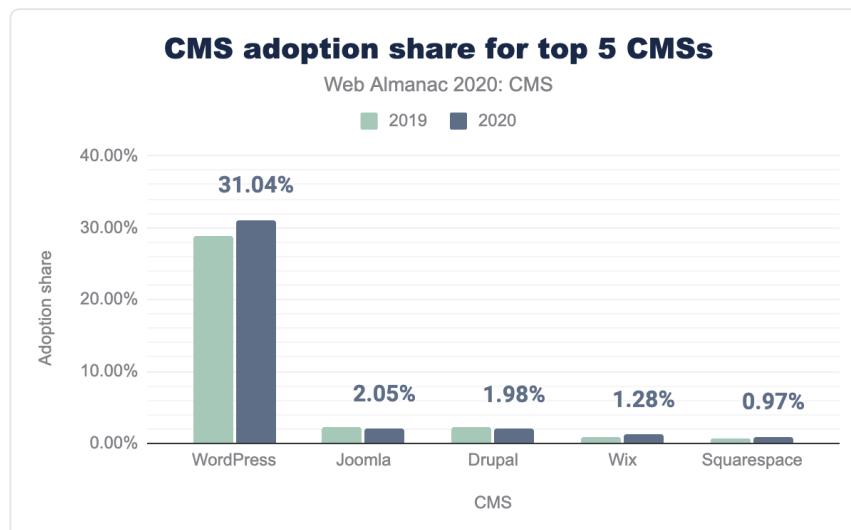


Figure 15.3. CMS adoption share for top 5 CMSs.

Joomla and Drupal have lost 8% and 10% of their adoption share respectively, whilst Wix and Squarespace have gained an extra 41% and 28% adoption share respectively. WordPress has gained an extra 7% adoption share in the last year, which is a larger absolute increase than the total share for Joomla, the next most popular CMS.

These numbers are broadly consistent when split across desktop and mobile:

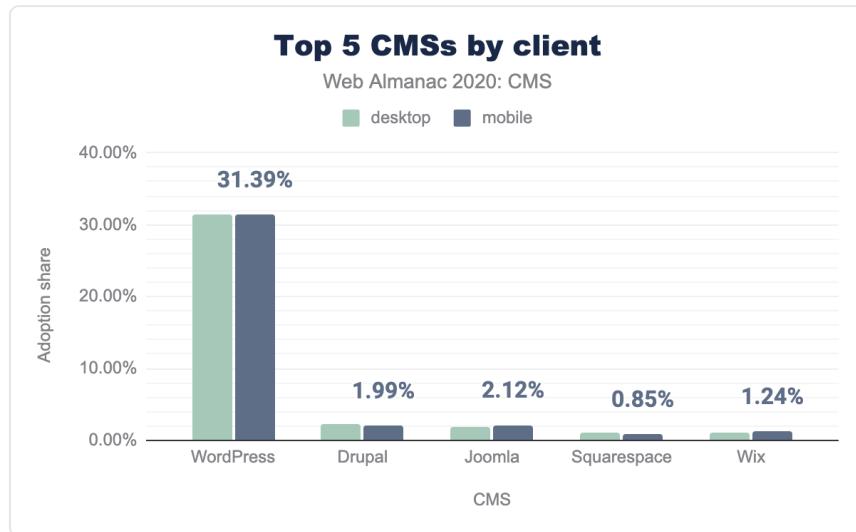


Figure 15.4. Top 5 CMSs by client.

For WordPress the numbers are very similar; for the other CMSs the difference is larger. Drupal and Squarespace have 16.7 and 26.3% more websites on desktop than mobile respectively, whilst Joomla and Wix have 7.5 and 15.2% more times on mobile than desktop.

The 0.1 to 1% adoption share category sees significantly more movement. These account for CMSs powering up to 50,000 websites.

CMS	2019	2020	% change
WordPress	28.91%	31.04%	7%
Joomla	2.24%	2.05%	-8%
Drupal	2.21%	1.98%	-10%
Wix	0.91%	1.28%	41%
Squarespace	0.76%	0.97%	28%
1C-Bitrix	0.55%	0.61%	10%
TYPO3 CMS	0.53%	0.52%	-2%
Weebly	0.39%	0.33%	-15%
Jimdo	0.28%	0.24%	-16%
Adobe Experience Manager	0.27%	0.23%	-14%
Duda		0.22%	
GoDaddy Website Builder		0.18%	
DNN	0.20%	0.16%	-19%
DataLife Engine	0.19%	0.16%	-12%
Tilda	0.08%	0.16%	100%
Liferay	0.12%	0.11%	-10%
Microsoft SharePoint	0.15%	0.11%	-25%
Kentico CMS	0.00%	0.11%	10819%
Contao	0.09%	0.09%	0%
Craft CMS	0.08%	0.09%	5%
MyWebsite		0.09%	
Concrete5	0.10%	0.09%	-12%

Figure 15.5. Relative % adoption of smaller CMSs (0.1% - 1% adoption share)

We see three new entrants here: Duda, GoDaddy Website Builder, and MyWebsite. Two, Tilda and Kentico CMS, have seen an adoption share change of over 100% in the last year. This "long

"tail" of CMSs cover a mix of open source and proprietary platforms and include everything from consumer-friendly to industry-specific. An incredible strength of the CMS platforms as a whole is one can get specialized software which powers every conceivable type of website.

When we look at CMS adoption share relative to other CMSs (thus excluding websites with no CMS), The dominance of WordPress becomes clear. The adoption share of websites with a CMS is 74.2%. With these numbers relative, Joomla, Drupal, Wix, and Squarespace receive higher adoption rates: 4.9%, 4.7%, 3.1%, and 2.3% respectively:

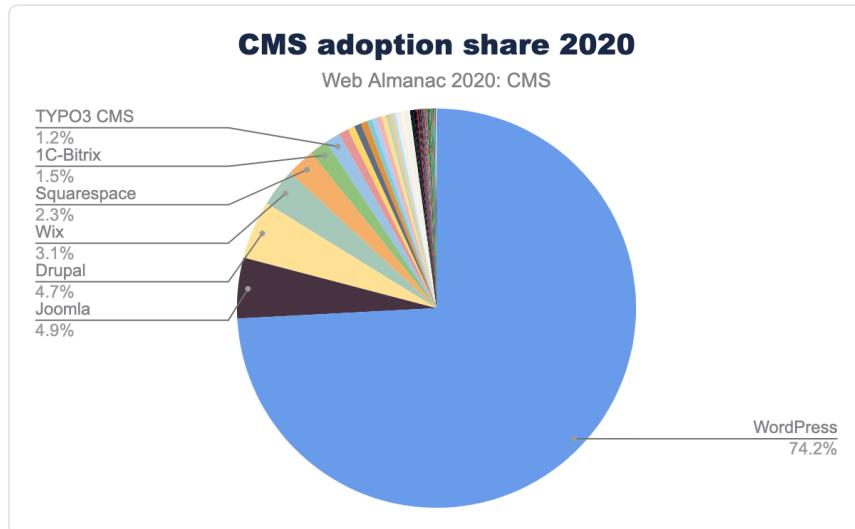


Figure 15.6. CMS adoption share 2020.

WordPress usage

WordPress dominates this space and thus deserves further discussion.

WordPress is an open source project with a mission to "democratize publishing". The CMS is free. While this is likely an important factor in its adoption share, the two next most popular CMSs—Joomla and Drupal—are also free. The WordPress community, contributors, and business ecosystem are likely the major differentiators.

A "core" WordPress community maintains the CMS and services requirements for additional functionality through custom services and products (themes and plugins). This community has an outsized impact, with a relatively small number of people maintaining both the CMS itself and providing the additional functionality which makes WordPress sufficiently powerful and flexible that it can service most types of website. This flexibility is important when explaining

the market share.

Deriving from this flexibility, WordPress also has a low barrier of entry for developers and site "builders" or "implementers". We see a virtuous cycle: flexible extensions offer ever-easier site building, which lets more and more users build ever-more-powerful sites with WordPress. This increase in users makes it more attractive for developers to create better and better extensions, furthering the cycle.

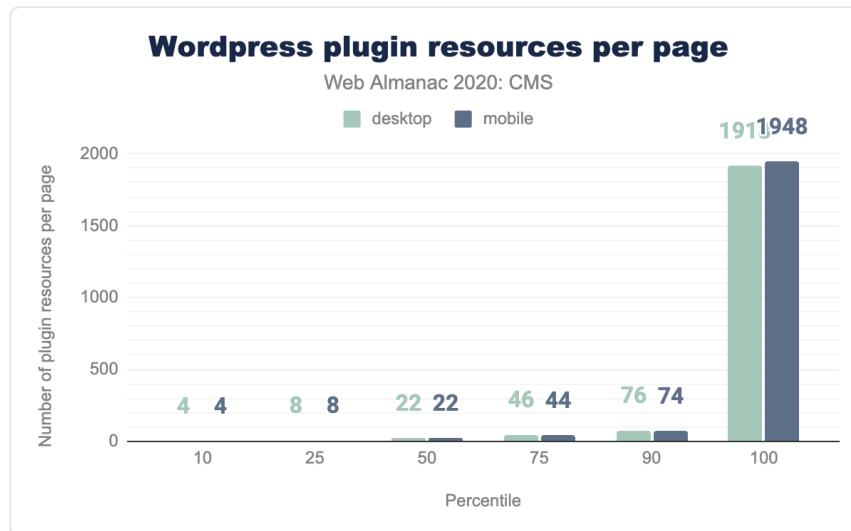


Figure 15.7. WordPress plugin resources per page.

We explored how WordPress sites use these extensions, which are typically WordPress plugins. The median WordPress site (on desktop and mobile) loads 22 plugin resources per page, with sites at the 90th percentile loading 76 and 74 resources per page on desktop and mobile respectively. At the 100th percentile this goes as high as 1918 and 1948 resources per page on desktop and mobile respectively. Whilst we can't compare this to other CMSs, it seems likely that WordPress's extension ecosystem is a major contributor to its high adoption rate.

WordPress's adoption share growth of 7.40% from 2019 to 2020 outstrips the overall increase in adoption of CMSs as a whole. This suggests WordPress has appeal significantly beyond the "average" CMS.

2020 has seen the impact of COVID-19. This may explain the increase in market share. Anecdotally, we can suggest that with many physical businesses closing permanently or temporarily, there has been increased demand for websites in general and WordPress as the largest CMS has benefited from this. Further research in the coming years will be required to ascertain the full impact.

With the adoption share of CMSs explored, let's now turn our attention to user experience.

CMS user experience

CMSs must offer a good user experience. With so much of the web relying on CMSs to serve pages, it is the responsibility of the CMS at the platform-level to ensure the user experience is good. Our aim is to shed light on real-world user experience when using CMS-powered websites.

To achieve this, we turn our analysis towards some user-perceived performance metrics, which are captured in the three Core Web Vitals metrics, as well as the Lighthouse scores in the SEO and Accessibility categories.

Chrome User Experience Report

In this section we take a look at three important factors provided by the Chrome User Experience Report, which can shed light on our understanding of how users are experiencing CMS-powered web pages in the wild:

- First Contentful Paint (FCP)
- First Input Delay (FID)
- Cumulative Layout Shift (CLS)

These metrics aim to cover the core elements which are indicative of a great web user experience. The Performance chapter will cover these in more detail, but here we are interested in looking at these metrics specifically in terms of CMSs. Let's review each of these in turn.

Largest Contentful Paint

Largest Contentful Paint (LCP) measures the point when the page's main content has likely loaded and thus the page is useful to the user. It does this by measuring the render time of the largest image or text block visible within the viewport.

This is different to First Contentful Pain (FCP), which measures from page load until content such as text or an image is first displayed. LCP is regarded as a good proxy for measuring when the main content of a page is loaded.

A "good" LCP is regarded as under 2.5 seconds. The average website on one of the top five CMSs does not have a good LCP. Only Drupal on desktop scores over 50% here. We see major

discrepancies between desktop and mobile scores: WordPress is fairly even at 33% on desktop and 25% on mobile, but Squarespace scores 37% on desktop and only 12% on mobile.

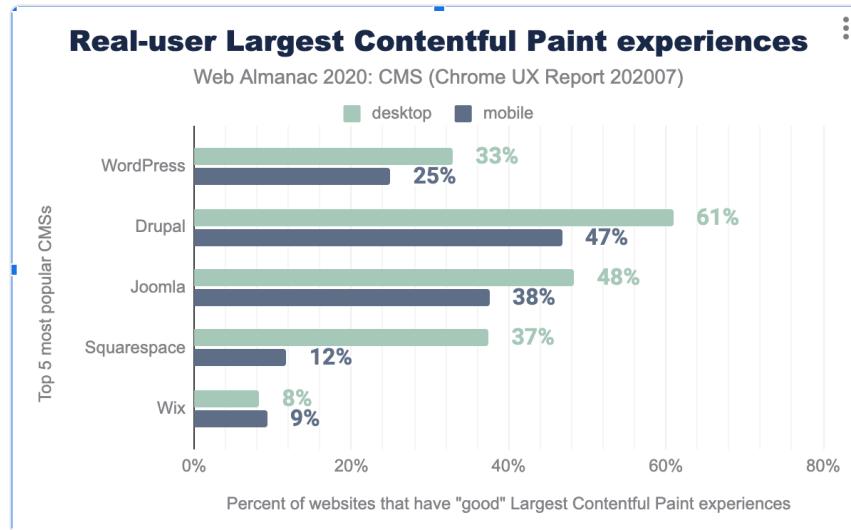


Figure 15.8. Real-user Largest Contentful Paint experiences.

Even though we'd love to see CMSs performing much better here, there are still some positive takeaways from these results. For one, the fact that 61% of Drupal websites have good LCP is especially notable because it's much better than the global distribution of 48% of websites having good LCP, according to the Chrome UX Report. For 1 in 3 or 4 WordPress websites to have good LCP is also kind of amazing, given the sheer magnitude of the number of WordPress websites. Wix does have some catching up to do, but it's encouraging to see that Wix engineers are actively working on fixing performance issues, so this will be something to keep an eye on over the years.

First Input Delay

First Input Delay (FID) measures the time from when a user first interacts with your site (i.e. when they click a link, tap on a button, or use a custom, JavaScript-powered control) to the time when the browser is actually able to respond to that interaction. A "fast" FID from a user's perspective would be immediate feedback from their actions on a site rather than a stalled experience. Any delay is a pain point and could correlate with interference from other aspects of the site loading when the user tries to interact with the site.

FID is very fast for the average CMS website on desktop—only Wix scores lower than 100%—and mixed on mobile. Most CMSs deliver mobile FID on an average site within a

reasonable range of the desktop score. For Wix the number of websites that have a good FID on mobile is nearly half the desktop total.

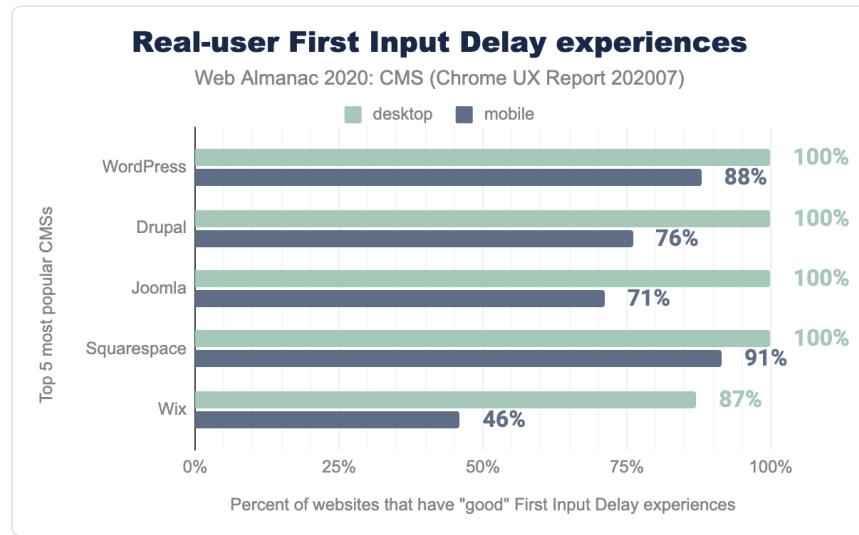


Figure 15.9. Real-user First Input Delay experiences.

The FID scores are generally good here, in contrast to the LCP scores. As suggested, the weight of individual pages on CMSs in addition to mobile connection quality or the lower performance of mobile devices relative to desktop, could play a role in the performance gaps that we see here affecting FID less.

There is a small margin of difference between the resources shipped to desktop and mobile versions of a website. Last year we noted that optimizing for the mobile experience was necessary. Average scores have increased on desktop and mobile, but further attention is required on mobile.

Cumulative Layout Shift

Cumulative Layout Shift (CLS) measures the instability of content on a web page after the first 500ms of user input, and after the first user input. This is important on mobile in particular, where the user will tap where they want to take an action—such as a search bar—only for the location to move as additional images, ads, or similar load.

A score of 0.1 or below is measured as "good", over 0.25 is "poor", and anything in between is "needs improvement".

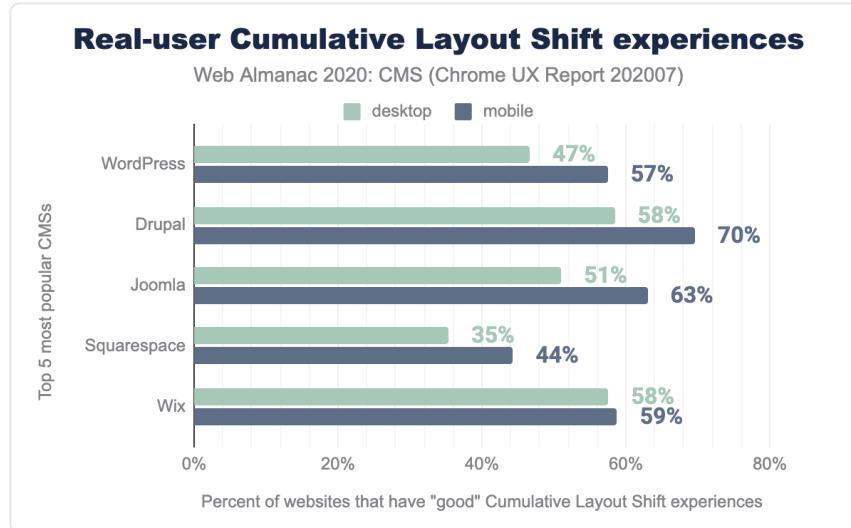


Figure 15.10. Real-user Cumulative Layout Shift experiences.

The top 5 CMSs could improve here. Only 50% of web pages loaded by a top 5 CMS have a "good" CLS experience, with this figure rising to 59% on mobile. Across all CMSs the average desktop score is 59% and average mobile score is 67%. This shows us all CMSs have work to do here, but the top 5 CMSs in particular need improvement.

Lighthouse scores

Lighthouse is an open-source, automated tool designed to help developers assess and improve the quality of their websites. One key aspect of the tool is that it provides a set of audits to assess the status of a website in terms of performance, accessibility, SEO, progressive web apps, and more. For this year's chapter, we looked at two specific audits categories: SEO and accessibility.

SEO

Search Engine Optimization (or SEO) is the practice of optimizing websites to make your website content more easily found in search engines. This is covered in more depth in our SEO chapter, but one part involves ensuring the site is coded in such a way to serve as much information to search engine crawlers to make it as easy as possible for them to show your site appropriately in search engine results. Compared to a custom created website, you would expect an CMS to provide good SEO capabilities, and the Lighthouse scores in this category show high marks:

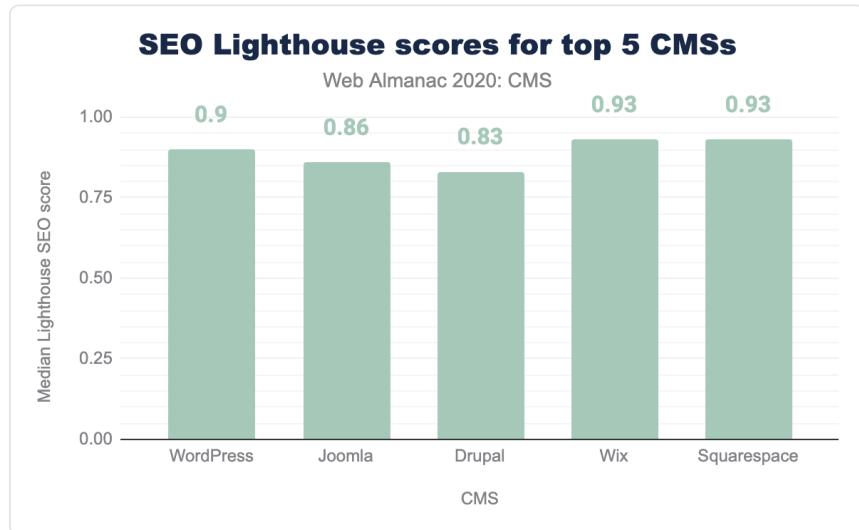


Figure 15.11. SEO Lighthouse scores for Top 5 CMSs.

All of the top 5 CMSs score highly here with median scores of 0.83 or above, with some reaching as high as 0.93. SEO can depend on the website owner making use of capabilities of a CMS but making those options easy to use in a CMS, and good defaults, can have big benefits for sites run on those CMSs.

Accessibility

An accessible website is a site designed and developed so that people with disabilities can use them. Web accessibility also benefits people without disabilities, such as those on slow internet connections. A full discussion can be seen here, and in our Accessibility chapter.

Lighthouse provides a set of accessibility audits and it returns a weighted average of all of them (see Scoring Details for a full list of how each audit is weighted).

Each accessibility audit is either a pass or a fail, but unlike other Lighthouse audits, a page doesn't get points for partially passing an accessibility audit. For example, if some elements have screen reader-friendly names, but others don't, that page gets a 0 for the screen reader-friendly-names audit.

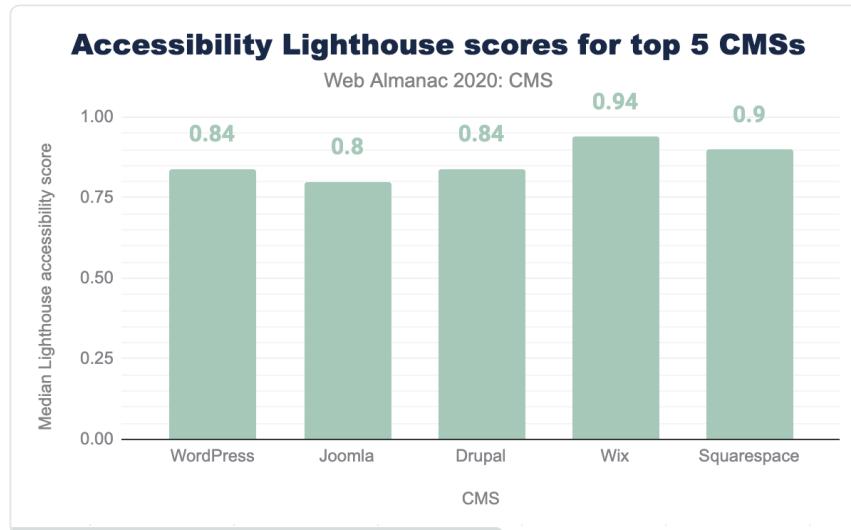


Figure 15.12. Accessibility Lighthouse scores for Top 5 CMSs.

The median Lighthouse accessibility score for the top 5 CMSs is all above 0.80. Across all CMSs, the average median Lighthouse score is 0.78, with a minimum of 0.44 and a maximum of 0.98. We thus see that the top 5 CMSs are better than average, with some better than others. Wix and Squarespace have the highest scores of the top 5. Possibly these platforms being proprietary helps here, as they're able to control the sites which are created more closely.

The bar should be higher here, though. An average score of 0.78 across all CMSs still leaves significant room for improvement, and the maximum score of 0.98 shows even the "best" CMS for accessibility compliance has room for improvement. Improving accessibility is essential and urgent work.

Environmental impact

This year we've sought to better understand the impact of CMSs on the environment. The information and communications technology (ICT) industry accounts for 2% of global carbon emissions, and data centers specifically account for 0.3% of global carbon emissions. This puts the ICT industry's carbon footprint equivalent to the aviation industry's emissions from fuel. We don't have data on the role of CMSs here, but with our research showing 42% of websites use a CMS, it is clear CMSs play an important role in the efficiency of websites and their impact on the environment.

Our research looked at the average CMS page weight in KB and mapped this to CO₂ emissions

using logic from carbonapi. This generated the following results, split by desktop and mobile:

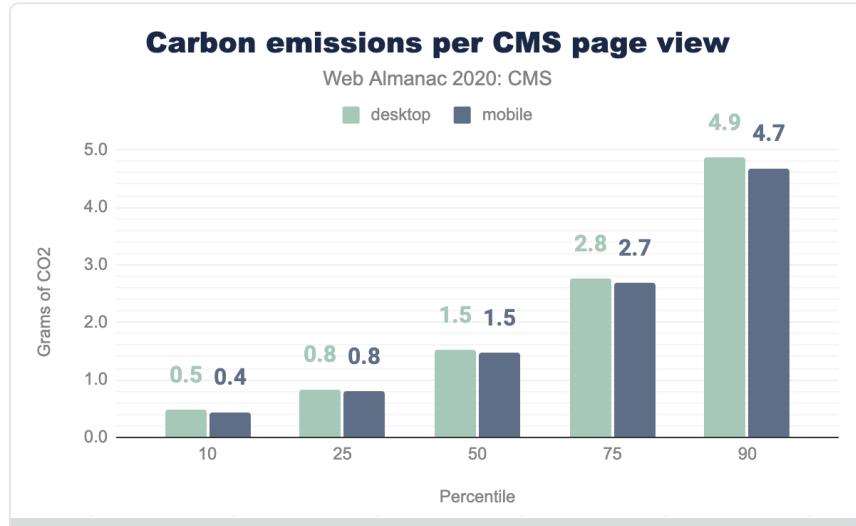


Figure 15.13. Carbon Emissions per CMS page view.

We found that the median CMS page load resulted in the transfer of 2.41 MB and thus the emission of 1.5g of CO₂. This was the same for desktop and mobile. The most efficient percentile of CMS web pages result in the generation of at least one third less CO₂, whilst the least efficient percentile of CMS web pages goes the other way: over one third less efficient than the median. The most efficient percentile of pages is approximately ten times more efficient than the least efficient percentile.

CMSs power every type of website, so this discrepancy is not surprising. CMSs can, however, influence at the platform-level the efficiency of websites they create.

Page weights are important here. The average desktop CMS web page loads 2.4 MB of HTML, CSS, JavaScript, media, etc. 10% of pages, however, load over 7 MB of this data. On mobile devices the average web page loads 0.1 MB fewer than on desktop, with at least this number being true across all percentiles:

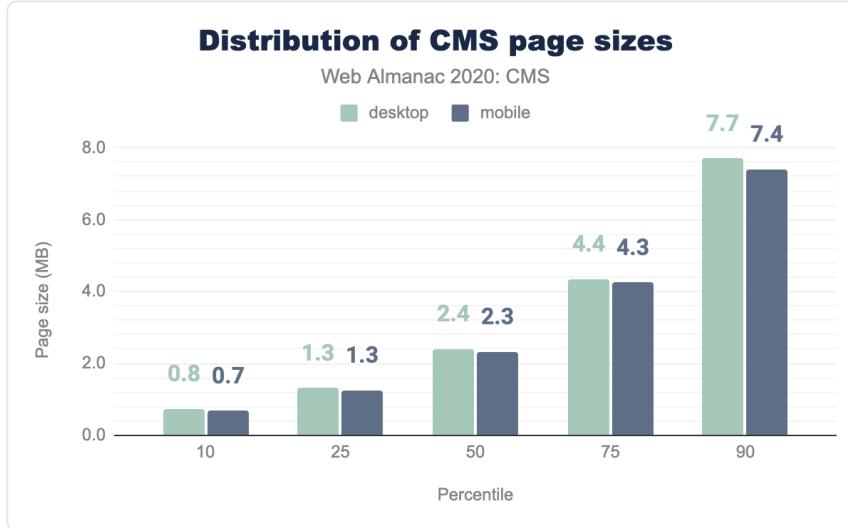


Figure 15.14. Distribution of CMS page sizes.

CMS often load third party resources, such as external images, videos, scripts, or stylesheets:

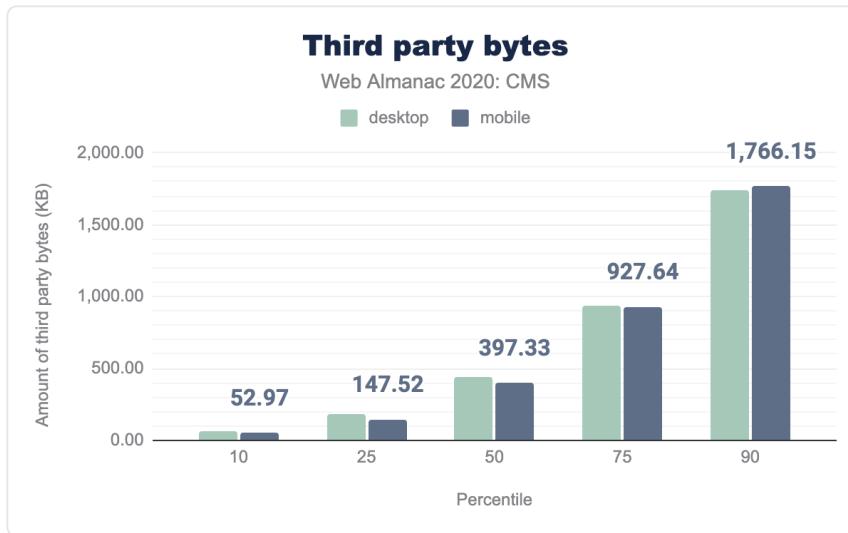


Figure 15.15. Third party bytes.

We find that the median desktop CMS page has 27 third-party requests with 436 KB of content, with the mobile equivalent generating 26 requests with 397 KB of content.

One of the main ways a CMS can influence its page load size is by supporting and encouraging the usage of more efficient formats. Images are behind only video in their contribution to page weight.

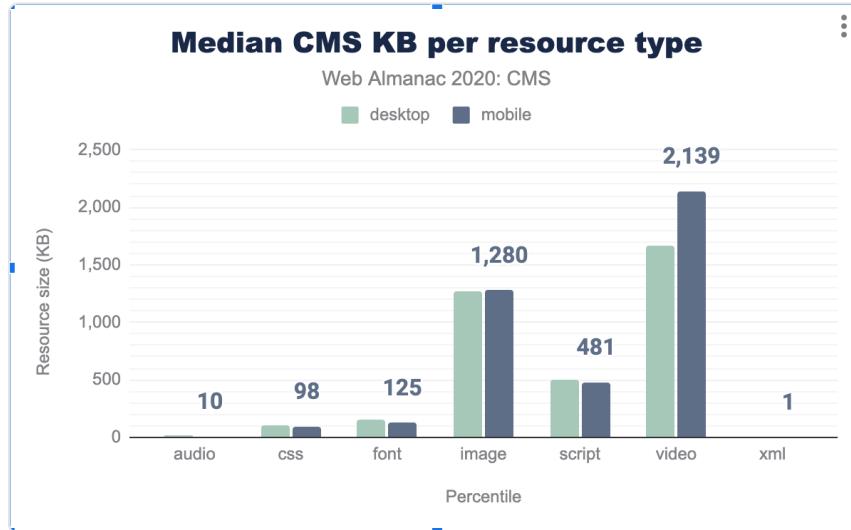


Figure 15.16. Median CMS KB per resource type.

Video contributes a larger percentage per resource type here. Making video more efficient, or other mechanisms such as the impact of stopping autoplay, are interesting areas for future research. Here our focus is on images. Popular image formats are JPEG, PNG, GIF, SVG, WebP, and ICO. Of these, WebP is the most efficient in most situations, with WebP lossless images 26% smaller than equivalent PNGs and 25-34% smaller than comparable JPGs. We see, however, that WebP is the second least popular image format across all CMS pages:

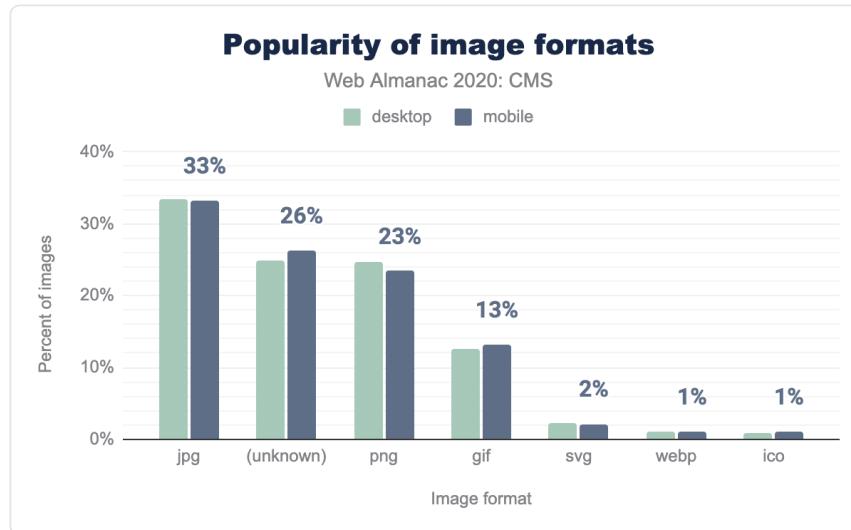


Figure 15.17. Popularity of image formats.

Of the top 5 CMSs, only Wix automatically converts and serves images in the WebP format. WordPress, Drupal, and Joomla support WebP with extensions, whilst at the time of writing Squarespace does not support WebP.

As we saw earlier, Wix had the lowest proportion of sites with a "good" LCP. While we know that Wix is making efficient use of image bytes in WebP, there are clearly other issues affecting its LCP performance beyond image formats that we aren't controlling for here. WebP is, however, a more efficient format and improved native support for the format by the most popular CMSs would be beneficial.

Image formats are one mechanism for making images more efficient. Other mechanisms such as "lazy loading" images would benefit from future research.

We're unable to fully answer the question of the impact of CMSs on the environment, but we are contributing to an answer. CMSs have a responsibility to take environmental impact seriously and decreasing the average page weight is important work.

Conclusion

CMSs have only gotten more important in the last year. They are essential for how content is created and consumed on the internet, and there are no signs that this will change in the foreseeable future. CMSs are set to become more important with each year passing.

We have reviewed the adoption of CMSs, user experience of websites created by these CMSs, and for the first time looked at the impact of CMSs on the environment. We have answered many questions here but leave further questions unanswered. Further research building on this chapter will be gratefully received. We have also highlighted some areas which need attention by the CMSs. We hope there will be progress to share in the 2021 report.

CMSs are vital for the success of the internet and open web. Let's work towards continued progress.

Author



Alex Denning

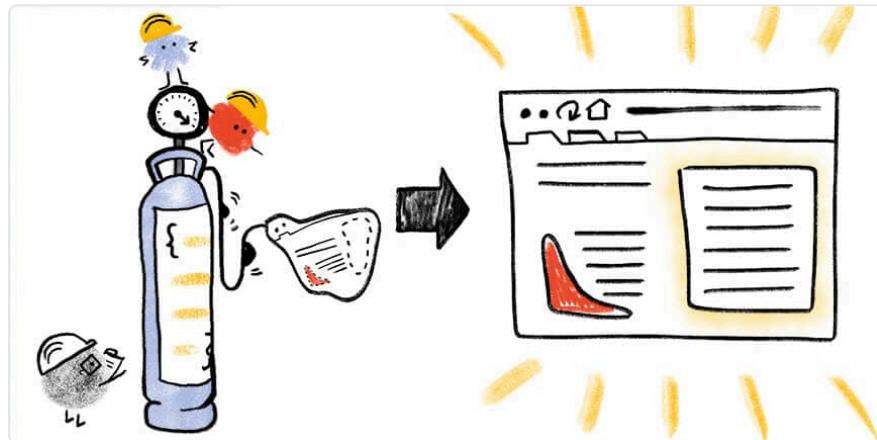
🐦 @AlexDenning ⚡ alexdenning 🌐 <https://getellipsis.com>

Alex Denning is the Founder of Ellipsis Marketing⁵⁰, a marketing agency for WordPress businesses. Alex is a WordPress Core Contributor and has helped organize WordCamp London⁵¹.

50. <https://getellipsis.com/>
51. <https://london.wordcamp.org/>

Part III Chapter 17

Jamstack



Written by Ahmad Awais

Reviewed by Maedah Batool and Nicolas Goutay

Analyzed by Artem Denysov and Brian Rinaldi

Introduction

Jamstack is a relatively new concept of an architecture designed to make the web faster, more secure, and easier to scale. It builds on many of the tools and workflows which developers love, and which maximizes productivity.

The core principles of Jamstack are pre-rendering your site pages and decoupling the frontend from the backend. It relies on the idea of delivering the frontend content hosted separately on a CDN provider that uses APIs (for example, a headless CMS) as its backend if any.

The HTTP Archive crawls millions of pages every month and runs them through a private instance of WebPageTest to store key information on every page crawled. You can learn more about this in our methodology page. In the context of Jamstack, HTTP Archive provides extensive information on the usage of the frameworks and CDNs for the entire web. This chapter consolidates and analyzes many of these trends.

The goals of this chapter are to estimate and analyze the growth of the Jamstack sites, the

performance of popular Jamstack frameworks, as well as an analysis of real user experience using the Core Web Vitals metrics.

It should be noted that our analysis is limited by those Jamstacks that make themselves easily identifiable using Wappalyzer. This means our data does not include some popular Jamstacks like Eleventy which make a deliberate choice to not make themselves identifiable. While we would ideally include all Jamstacks, we believe there is still plenty of value in analyzing the significant data we do have.

Adoption of Jamstack

Our analysis throughout this work looks at desktop and mobile websites. The vast majority of URLs we looked at are in both datasets, but some URLs are only accessed by desktop or mobile devices. This can cause small divergences in the data, and we thus look at desktop and mobile results separately.

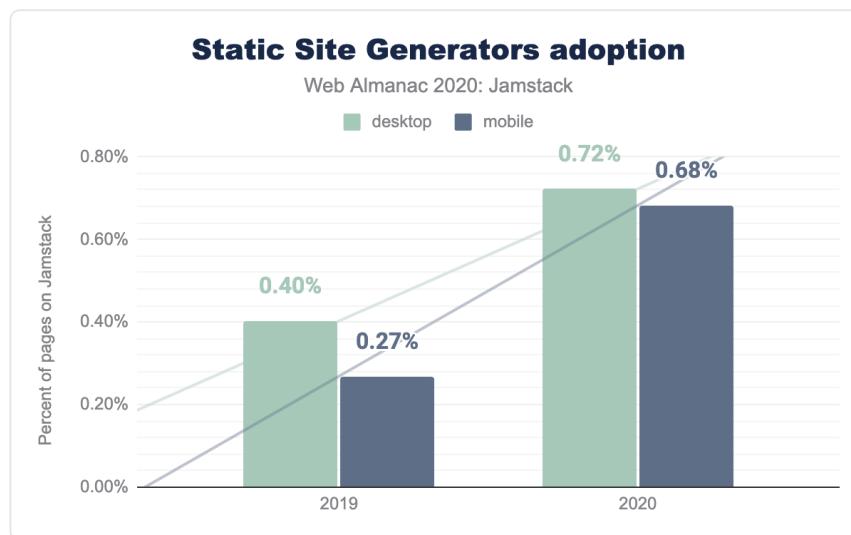


Figure 17.1. Jamstack adoption trend.

More than 0.7% of web pages are powered by Jamstack and breaks down to 0.72% on desktop, up from 0.40% in 2019, and 0.68% on mobile, up from 0.27% in 2019.

Year	Desktop	Mobile
2019	0.40%	0.27%
2020	0.72%	0.68%
% Change	79%	154%

Figure 17.2. Jamstack adoption statistics.

The increase in desktop web pages powered by a Jamstack framework is 79% from last year. On mobile, this increase is almost two folds, at 154%. This is a significant growth from 2019, especially for mobile pages. We believe this is a sign of the steady growth of the Jamstack community.

Jamstack frameworks

Our analysis counted 12 separate Jamstack frameworks. Only five frameworks had more than 1% share: Next.js, Gatsby, Hugo, Jekyll are the top contenders for the Jamstack market share.

In 2020, most of the Jamstack market share seems distributed between the top four frameworks. Interestingly, Next.js has 72.5% usage share. This is nearly five times the share of the next most popular Jamstack framework, Gatsby at just under 15%!

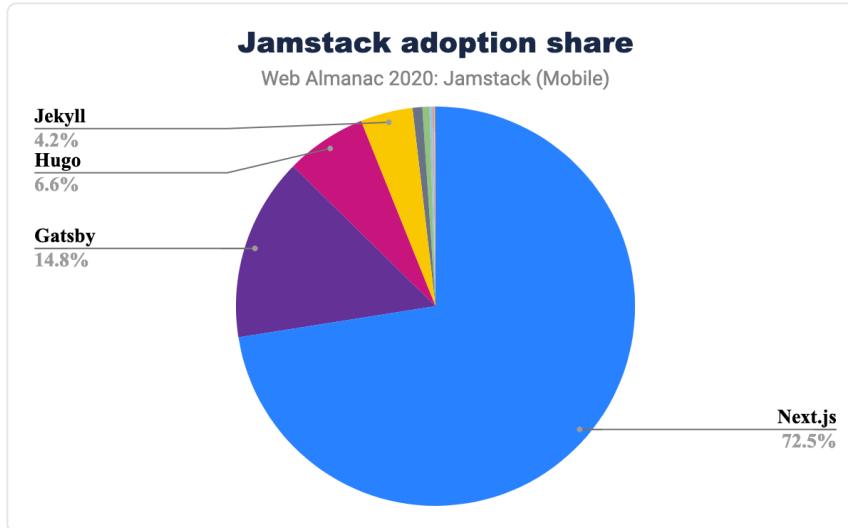


Figure 17.3. Jamstack adoption share pie chart 2020.

Framework adoption changes

Looking at the year on year growth, we see that Next.js has increased its lead over its competitors in the last year:

Jamstack	2019	2020	% change
Next.js	61.06%	72.51%	19%
Gatsby	15.87%	14.84%	-6%
Hugo	12.12%	6.56%	-46%
Jekyll	7.92%	4.24%	-46%
Hexo	1.48%	0.79%	-46%
Gridsome	0.25%	0.57%	133%
Octopress	0.78%	0.24%	-69%
Pelican	0.39%	0.14%	-65%
VuePress		0.06%	
Phenomic	0.13%	0.03%	-78%
Saber		0.01%	
Cecil	0.01%		

Figure 17.4. Relative % adoption of Jamstack frameworks

And concentrating on the top 5 Jamstacks further shows Next.js's lead:

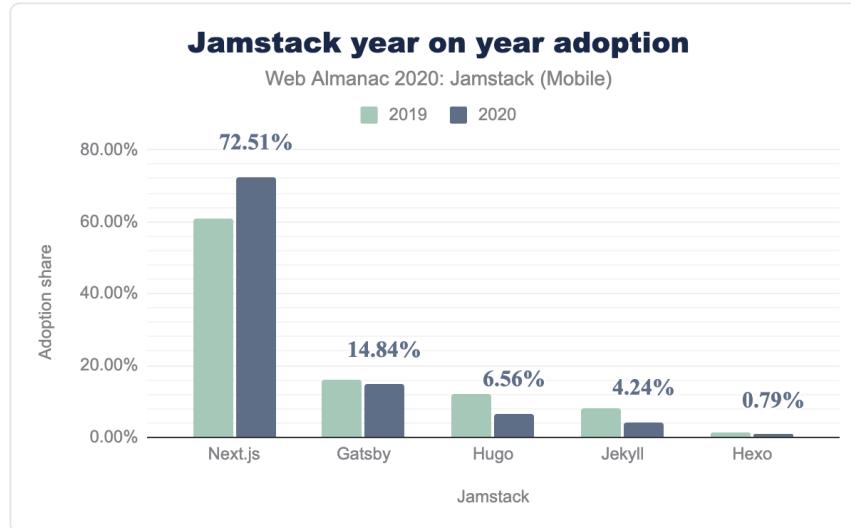


Figure 17.5. Jamstack adoption share year on year.

It's worth noting here the fact that Next.js websites include a mix of both Static Site Generated (SSG) pages and Server-Side Rendered (SSR) pages. This is due to the lack of our ability to measure them separately. This means that the analysis may include sites that are mostly or partially server-rendered, meaning they do not fall under the traditional definition of a Jamstack site. Nonetheless, it appears that this hybrid nature of Next.js gives it a competitive advantage over other frameworks hence making it more popular.

Environmental impact

This year we have sought to better understand the impact of Jamstack sites on the environment. The information and communications technology (ICT) industry accounts for 2% of global carbon emissions, and data centers specifically account for 0.3% of global carbon emissions. This puts the ICT industry's carbon footprint equivalent to the aviation industry's emissions from fuel.

Jamstack is often credited for being mindful of performance. In the next section, we look into the carbon emissions of Jamstack websites.

Page weight

Our research looked at the average Jamstack page weight in KB and mapped this to CO2

emissions using logic from the Carbon API. This generated the following results, split by desktop and mobile:

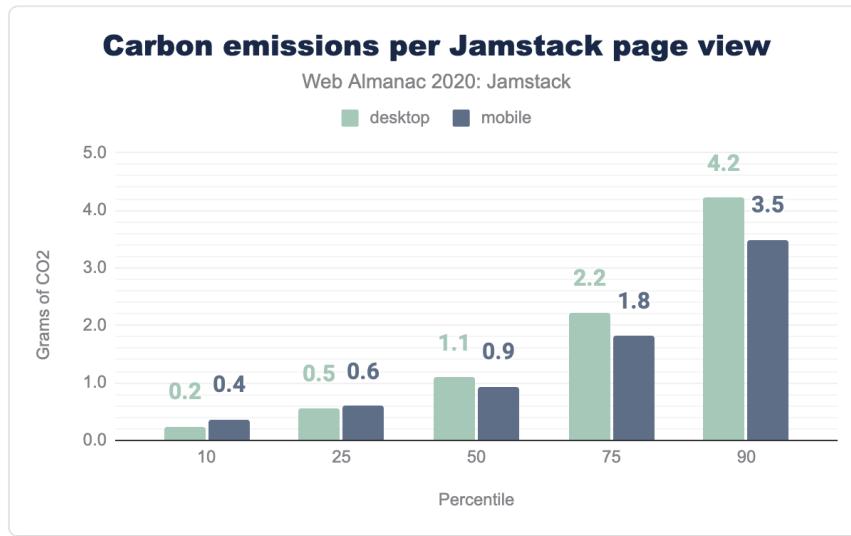


Figure 17.6. Carbon Emissions per Jamstack page view.

We found that the median Jamstack page load resulted in the transfer of 1.5 MB of various assets and thus the emission of 1.1 grams of CO₂ for desktop and 0.9 grams for mobile. This was the same for desktop and mobile. The most efficient percentile of Jamstack web pages result in the generation of at least one third less CO₂ than the median, whilst the least efficient percentile of Jamstack web pages goes the other way, generating around four times more.

Page weights are important here. The average desktop Jamstack web page loads 1.5 MB of video, image, script, font, CSS, and audio data. 10% of pages, however, load over 4 MB of this data. On mobile devices, the average web page loads 0.7 MB fewer than on desktop, a fact consistent across all percentiles.

Image formats

Popular image formats are PNG, JPG, GIF, SVG, WebP, and ICO. Of these, WebP is the most efficient in most situations, with WebP lossless images 26% smaller than equivalent PNGs and 25-34% smaller than comparable JPGs. We see, however, that WebP is the second least popular image format across all Jamstack pages, where PNG is the most popular both for mobile and desktop. Only slightly less popular is JPG whereas GIF is almost 20% of all the images used on Jamstack sites. An interesting discovery is SVG which is almost twice as popular on mobile sites as desktop sites.

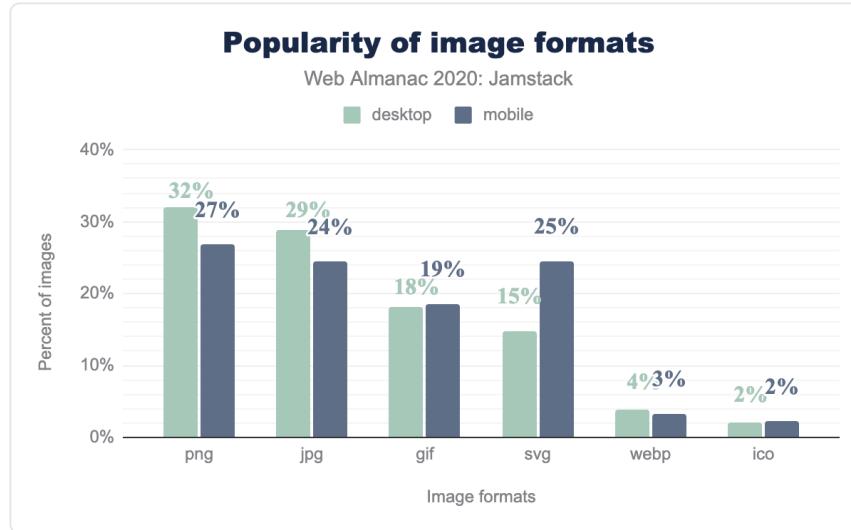


Figure 17.7. Popularity of image formats.

Third-party bytes

Jamstack sites, like most websites, often load third-party resources, such as external images, videos, scripts, or stylesheets:

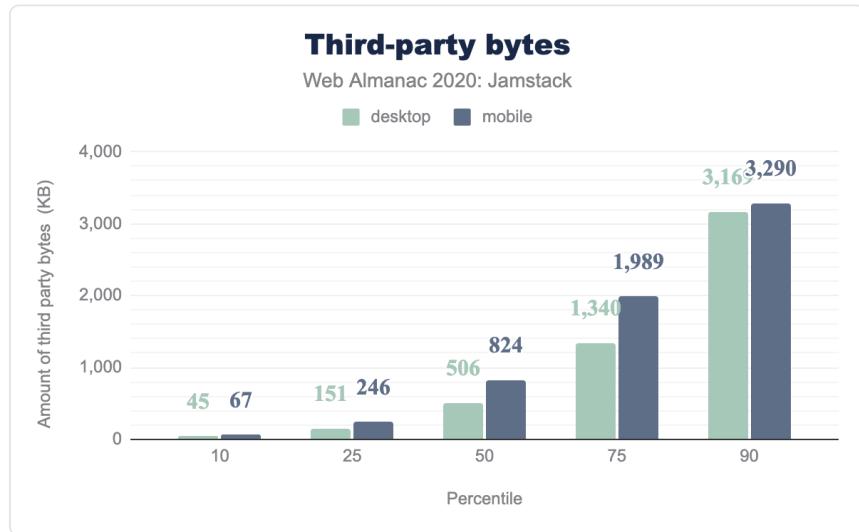


Figure 17.8. Third party bytes.

We find that the median desktop Jamstack page has 28 third-party requests with 506 KB of content, with the mobile equivalent generating 46 requests with 824 KB of content. Whereas 10% of the desktop sites have 132 requests with 3.1MB of content which is only superseded by 168 requests on mobile with 3.2MB of content.

User experience

Jamstack websites are often said to offer a good user experience. It's what the entire concept of separating the frontend from the backend and hosting it on the CDN edge is all about. We aim to shed light on real-world user experience when using Jamstack websites using the recently launched Core Web Vitals.

The Core Web Vitals are three important factors which can shed light on our understanding of how users are experiencing Jamstack pages in the wild:

- Largest Contentful Paint (LCP)
- First Input Delay (FID)
- Cumulative Layout Shift (CLS)

These metrics aim to cover the core elements which are indicative of a great web user experience. Let's we take a look at the real-world Core Web Vitals statistics of the top-five Jamstack frameworks.

Largest Contentful Paint

Largest Contentful Paint (LCP) measures the point when the page's main content has likely loaded and thus the page is useful to the user. It does this by measuring the render time of the largest image or text block visible within the viewport.

This is different from First Contentful Paint (FCP), which measures from page load until content such as text or an image is first displayed. LCP is regarded as a good proxy for measuring when the main content of a page is loaded.

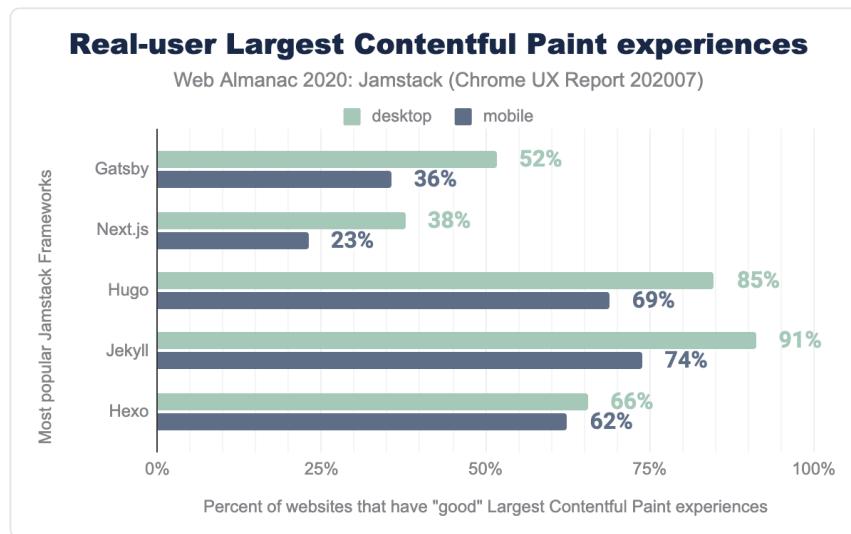


Figure 17.9. Real-user Largest Contentful Paint experiences.

A "good" LCP is regarded as under 2.5 seconds. Hugo, Jekyll, and Hexo have impressive LCP scores all above 50% with Jekyll and Hugo on desktop at 91% and 85% on desktop respectively. Gatsby and Next.js sites lagged – scoring 52% and 38% respectively on desktop, and 36% and 23% on mobile.

This might be attributed to the fact that most of the sites built with Next.js and Gatsby have complex layouts and high page weights, in comparison with Hugo and Jekyll which are primarily used to produce static content sites with fewer or no dynamic parts. For what it's worth, you don't have to use React or any other JavaScript framework with Hugo or Jekyll.

As we explored in the section above, high page weights can have a possible impact on the environment. However, this also affects LCP performance, which is either very good or generally bad depending on the Jamstack framework. This can have an impact on the real user experience as well.

First Input Delay

First Input Delay (FID) measures the time from when a user first interacts with your site (i.e. when they click a link, tap on a button, or use a custom, JavaScript-powered control) to the time when the browser is actually able to respond to that interaction.

A "fast" FID from a user's perspective would provide immediate feedback from their actions on a site rather than a stalled experience. This delay is a pain point and could correlate with interference from other aspects of the site loading when the user tries to interact with the site.

FID is extremely fast for the average Jamstack website on desktop – most popular frameworks score 100% – and above 80% on mobile.

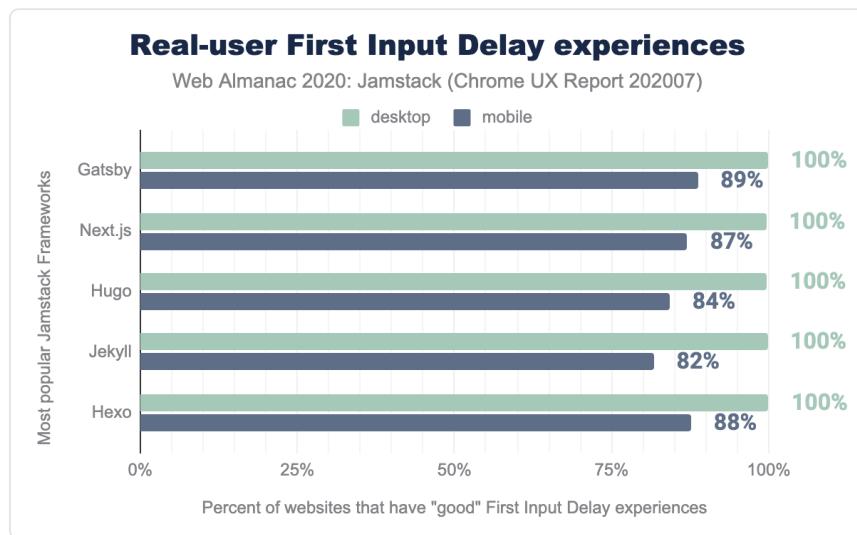


Figure 17.10. Real-user First Input Delay experiences.

There is a small margin of difference between the resources shipped to desktop and mobile versions of a website. The FID scores are generally very good here, but it is interesting this does not translate to similar LCP scores. As suggested, the weight of individual pages on Jamstack sites in addition to mobile connection quality could play a role in the performance gaps that we see here.

Cumulative Layout Shift

Cumulative Layout Shift (CLS) measures the instability of content on a web page within the first 500ms of user input. CLS measures any layout changes which happen after user input. This is

important on mobile in particular, where the user will tap where they want to take an action – such as a search bar – only for the location to move as additional images, ads, or similar load.

A score of 0.1 or below is good, over 0.25 is poor, and anything in between needs improvement.

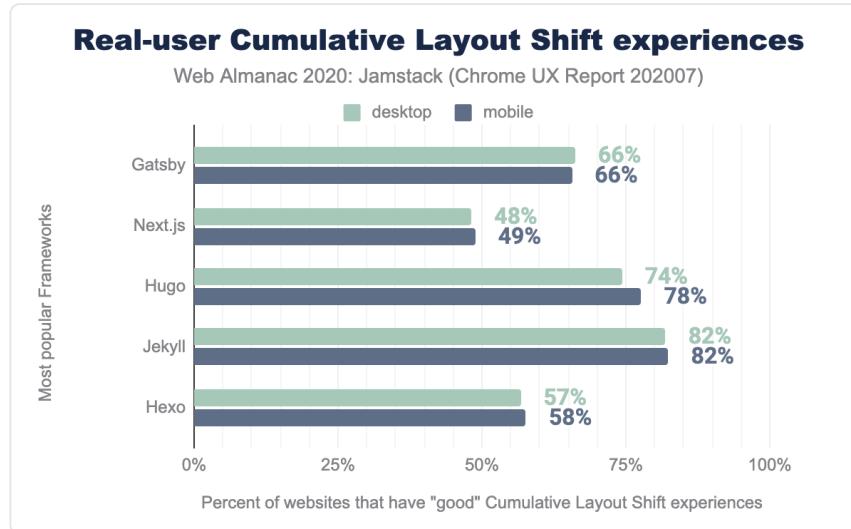


Figure 17.11. Real-user Cumulative Layout Shift experiences.

The top-five Jamstack frameworks do OK here. About 65% of web pages loaded by top-five Jamstack frameworks have a "good" CLS experience, with this figure rising to 82% on mobile. Across all the average desktop and mobile score is 65%. Next.js is barely reaching 50% which is the lowest of all and has work to do here. Educating developers and documenting how to avoid bad CLS scores can go a long way.

Conclusion

Jamstack, both as a concept and a stack, has picked up importance in the last year. Stats suggest almost twice as many Jamstack sites exist now than in 2019. Developers enjoy a better development experience by separating the frontend from the backend (a headless CMS, serverless functions, or third-party services). But what about the real-user experience of browsing Jamstack sites?

We've reviewed the adoption of Jamstack, user experience of websites created by these Jamstack frameworks, and for the first time looked at the impact of Jamstack on the environment. We have answered many questions here but leave further questions unanswered.

There are frameworks like Eleventy which we weren't able to measure or analyze since there is no pattern available to determine the usage of such frameworks, which has an impact on the data presented here. Next.js dominates usage and offers both Static Site Generation and Server-Side Rendering, separating the two in this data is nearly impossible since it also offers incremental Static Generation. Further research building on this chapter will be gratefully received.

Moreover, we have highlighted some areas which need attention from the Jamstack community. We hope there will be progress to share in the 2021 report. Different Jamstack frameworks can start to document how to improve real user experience by looking at Core Web Vitals.

Vercel, one of the CDNs meant to host Jamstack sites, has built an analytics offering called Real User Experience Score. While other performance measuring tools like Lighthouse estimate your user's experience by running a simulation in a lab, Vercel's Real Experience Score is calculated using real data points collected from the devices of the actual users of your application.

It is probably worth noting here that Vercel created and maintains Next.js, since Next.js had low LCP scores. This new offering could mean we can hope to see a marked improvement in those scores next year. This would be extremely helpful information for users and developers alike.

Jamstack frameworks are improving the developer experience of building sites. Let's work towards continued progress for improving the real-user experience of browsing Jamstack sites.

Author



Ahmad Awais

Twitter: @MrAhmadAwais GitHub: ahmadawais Website: <https://AhmadAwais.com>

Ahmad Awais is an award-winning open-source engineer, Google Developers Expert Dev Advocate, Node.js Community Committee Outreach Lead, WordPress Core Dev, and VP of Engineering DevRel at WGA. He has authored various open-source software tools used by millions of developers worldwide. Like his Shades of Purple⁵² code-theme or projects like the corona-cli⁵³. Awais loves to teach. Over 20,000 developers are learning from his courses⁵⁴ i.e. Node CLI⁵⁵, VSCode.pro⁵⁶, and Next.js Beginner⁵⁷. Awais received FOSS community leadership recognition as one of the 12 featured GitHub Stars⁵⁸. He is a member of the Smashing Magazine Experts Panel; featured & published author at CSS-Tricks, Tuts+, Scotch.io, SitePoint. You can mostly find him on Twitter @MrAhmadAwais where he tweets his #OneDevMinute⁵⁹ developer tips.

52. <https://shadesofpurple.pro/more>
53. <https://github.com/AhmadAwais/corona-cli>
54. <https://ahmadawais.com/courses>
55. <https://nodecli.com/>
56. <https://vscode.pro/>
57. <https://nextjsbeginner.com/>
58. <https://ahmadawais.com/github-stars/>
59. <https://awais.dev/odmt>

Part IV Chapter 18

Page Weight [UNEDITED]



Written by Henri Helvetica

Reviewed by Paul Calvano

Analyzed by Paul Calvano

Introduction

Page weight is one of the simpler metrics available. Much like stepping on a human scale to get a sense of your personal weight (well, mass really, but you get it), loading a page will provide a sense of the amount of resources collected and requested. But as the web and web pages have matured and grown, so have associated metrics — such as page weight. It can affect a page's performance much like personal weight (mass) can do the same. This chapter will take a deeper dive and peel back the layers of web pages, and see what it is that constitutes a page's weight at the possible detriment of the end user: you, I, us.

#PageWeightStillMatters

#PageWeightStillMatters would almost imply that it didn't or ever mattered. It might not have mattered when text based Craigslist launched. But 25 years ago when it was founded, Mosaic

1.0 also launched the same year, and Waterfalls by TLC was a top hit. The web matured as did resources. It was just a few years back when the twitterverse was tied up discussing how the average size of web pages now equaled the size of the original doom. Many of us mused about what the size the page could become in time, including our very own Tammy Everts, but the reality is startling. A page sits @ ~4 MB and 3.7 MB, desktop/mobile respectively, at the 75th percentile, and a shocking 7.4 MB and 6.7 MB at the 90th percentile. There are multitudes of implications in having such heavy pages, like the likelihood of poor user experience due to unreliable networks. Today, despite lessons learned a decade ago, we are experiencing variations of the same challenges: despite having slightly better networks, we are working with much larger resources.

Bandwidth

In 2016, when asked to explain why I asked an Australian tourist who is delighted with UK Internet, Google's Ilya Grigorik had two words: physics damn'it! (whoops, that's three). The point was simple: though you might benefit from increased bandwidth, the laws of physics still prevail. An Australian is unable to escape laws of latency. In the best case scenario, at home in Sydney, this Australian was experiencing enough latency that his Internet was at times perceived as unresponsive.

Now, imagine that the same Australian, knowing that at the 75th percentile, his page is making about 108 requests (more on that later), and we still have no idea of the network protocol, the resources being requested, the level of compression or optimization. You can pursue the H/2 and compression chapters for more information on the life of a modern request.

Assets

In 25 years of modern browsing, the assets and resources have mostly not changed, other than the amount. The HTTP archive modus operandi is "how the web was built", and that was mostly done with HTML, CSS, JavaScript and finally images.

Prior to 1995, the web's page weight was mostly predictable and manageable. But with RFC 1866, which introduced HTML 2.0, responsible for the introduction of inline images via the IMG tag, page weight would make a dramatic increase, all for the good of web development (adding images was seen as a positive experiment).

For the most part, the rule of thumb has been that images would make up the majority of page weight. It was certainly the case and a concern when in-line images were added to the web then, and remains the case today. In a separate scenario, as image data will be the greatest source of page weight, it will also be the greatest source of page weight savings (more on that later). This will be achieved from ensuring that the images are sized properly, but also making

sure that the images are at the optimization sweet spot - finding the best balance of quality and file size.

Although JavaScript is on average the 2nd most abundant resource on a page, we tend to have more opportunities in working with that file type: from bundling, compression and minification to name a few.

Intricate and interactive

The web's journey from the plain, near pedagogical platform, to the innovative, intricate and highly interactive apps it has become, the rudimentary page weight metric hid a bigger story: a ratatouille of resources, each affecting modern metrics, in turn affecting user experience.

Whenever we talk about interactivity, we are talking almost exclusively about JavaScript. Now, though we are not here to discuss interactivity in any depth, we know there are metrics which are focused and dependent on JavaScript content and execution. So weightier the JavaScript, likely to have a greater impact on interactivity metrics (time to interactive, total blocking time). We have the JavaScript chapter that dives a pinch more.

Analysis

As we post and parse the statistical results, just a reminder that the data is based on transfer sizes. Added, we are employing decompressed sizes in this analysis when possible.

Page weight

Let's look at the classic page weight, on both desktop and mobile. The deltas are mostly due to a few less resources transferred on mobile, a likely pinch of media management, but you can see below that at the median, the differences are not that significant between the two clients. We are highlighting the mobile results.

We can however surmise from this the following: we are closing in on 7 MB of page weight on mobile and 7.5 MB on desktop at the 90th percentile. The data is following an age old trend: growth in page weight is on the upward trajectory yet again, from the previous year. Popping the hood, we can see how things look at the median and average for each resource. One thing again remains: images are the dominant resource and JavaScript is the second most abundant, though a far second.



Figure 18.1. Distribution of total bytes per page.

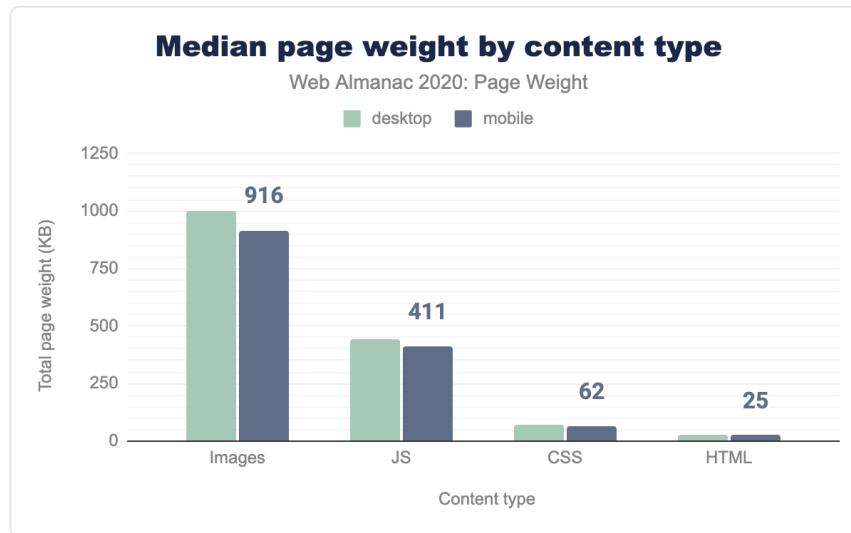


Figure 18.2. Median bytes per page by content type.

Requests

We have an old adage: the quickest request is the one never made. Dare we then say: the smallest resource is one never requested.

at the request level, much is the same. The weightiest resources are making the most requests. The request distribution will show that the difference between desktop and mobile is not so significant, with desktop leading the way. Something worth noting: the median request on desktop at this time is the same as last year later (74), yet the page weight has ticked up (+122kb). A simple observation, but one which confirms the trajectory we've seen over the years.



Figure 18.3. Distribution of requests per page.

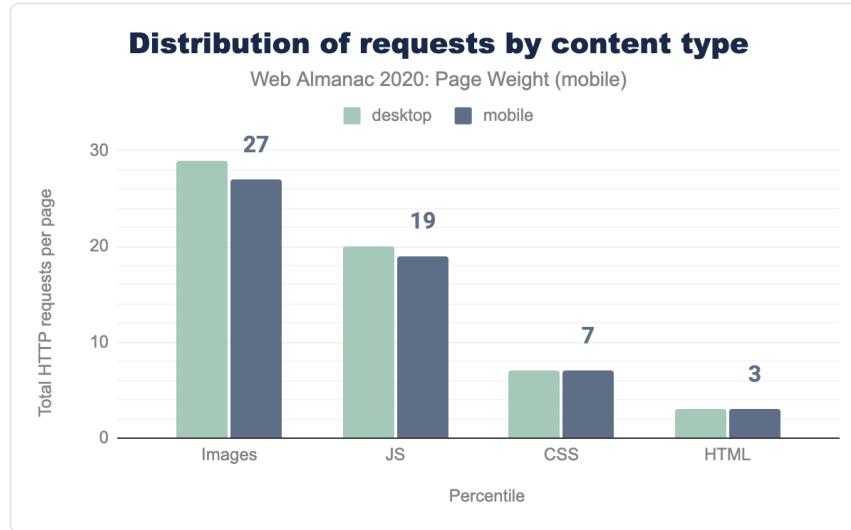


Figure 18.4. Median number of requests per mobile page by content type.

File formats

We know that images are a great source of page weight. This graphic below shows us the top sources of image weight and the weight distribution. Top 3: JPG, PNG and WebP. So not only is the JPG the most popular image format, it also tends to be the largest by size as well - even larger than a lossless format like the PNG. But as we noticed last year, that has to do with the predominant use case for the PNG, which seems to be icons and logos.

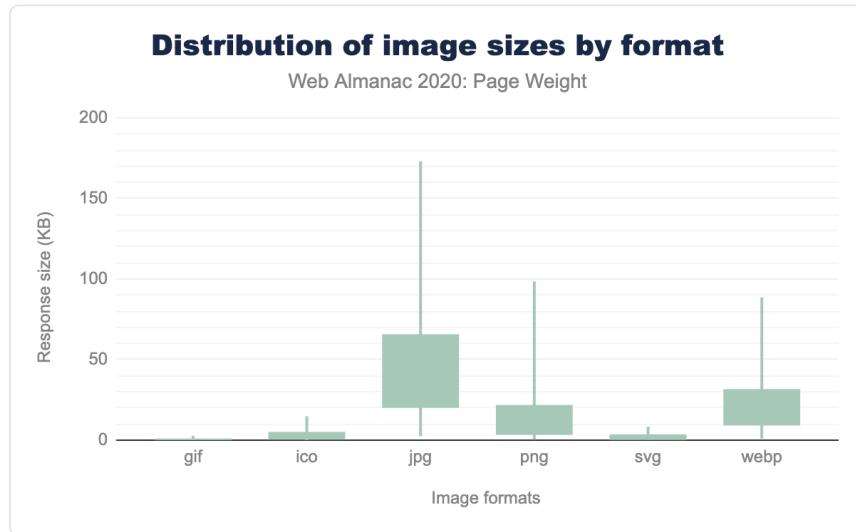


Figure 18.5. Distribution of image sizes by format.

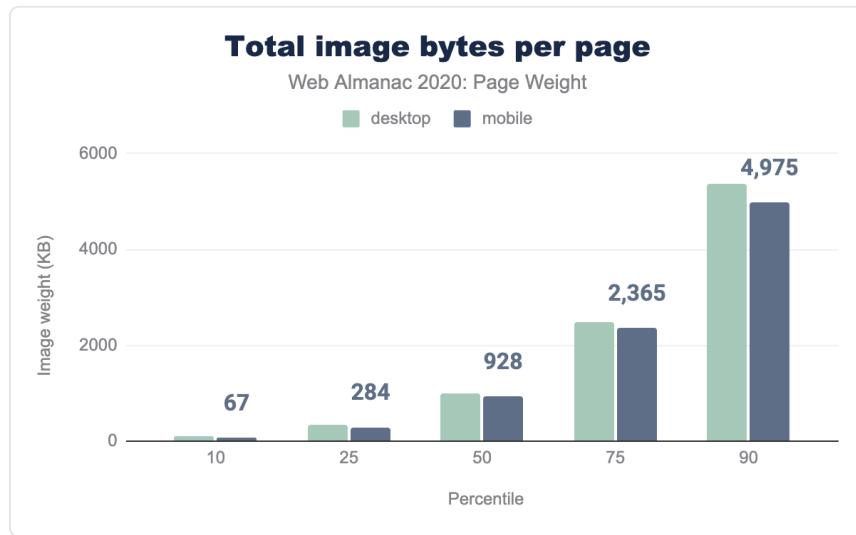


Figure 18.6. Distribution of image response sizes per page.

COVID-19

2020 has been the most demanding of any year in Internet history. This is based on self

reporting by telecom companies all over the globe. YouTube, Netflix, gaming console manufacturers and many more were asked to throttle their networks due to anticipated bandwidth demands of COVID-19 and the stay at home orders. There are now new suspects creating demands on the networks: we are now working from home, teleconferencing from home, and now schooling from home as well. In the midst of this crisis some government organizations have moved forward to optimize all aspects of the site and redesign and or updates. Two such examples of being ca.gov ([link](#)) and gov.uk. In these times, COVID-19 has certified the Internet as an essential service, and being able to access crucial and life-saving information, Must be as friction free as possible, which includes a manageable page weight via discipline delivery of data.

If we have been married to the Internet, COVID-19 has forced us to renew our vows. Assuring that content is delivered as efficiently as possible over the Internet, page weight must be kept at the forefront at all times.

A not so distant future

We have watched for 25 years page weight grow steadily. It might have been one of the greatest stock investments — had it been one. But this is the web. And, we are trying to manage data, requests, file size and ultimately page weight. We have just combed over data, seeing how images are the greatest source of weight. This means, it will also be our greatest source of savings. 2020 was a pivotal year, a possible inflection point for HTTP Archive tracking of web data. 2020 marked the year the modern format webp was finally adopted by Safari, making this format finally supported by all browsers across the board. This means that the format could comfortably be used with little to no fall back. The most important point? The potential for significant page weight savings is unavoidable — at a possible 30%. Even more interesting is the idea of a more modern format: avif. This format has burst onto the scene with enough support today for approximately 70% browser market share, creating a scenario for small image file sizes - even smaller than webp. And lastly, and possibly most distant: media queries level 5, ‘prefers-reduced-data’. Though in very early draft, this media feature will be used to detect if a user may have a preference for variant resources in data sensitive situations

Looking at the crystal ball, the third installment of the Web Almanac and the page weight chapter could have a much different look in 2021. The big technological and engineering investments into images, might finally provide the diminishing returns we have been looking for.

Conclusion

It's of no surprise that web pages have generally kept growing. We have been feeding more

resources down the wire to create richer experiences, more engaging interactivity, more stunning visuals through more powerful imagery. We have created these applications at the cost of data overages and user experiences. But as we move forward and keep pushing the web to places we had never anticipated, we are also making additional advances in engineering, as mentioned earlier. We may begin to see a drop in page weight as early as next year, as modern raster image formats see more adoption, we start to manage JavaScript more efficiently, and deliver the data down the wire with the discipline that users demand.

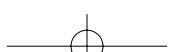
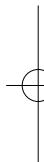
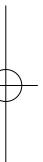
Author



Henri Helvetica

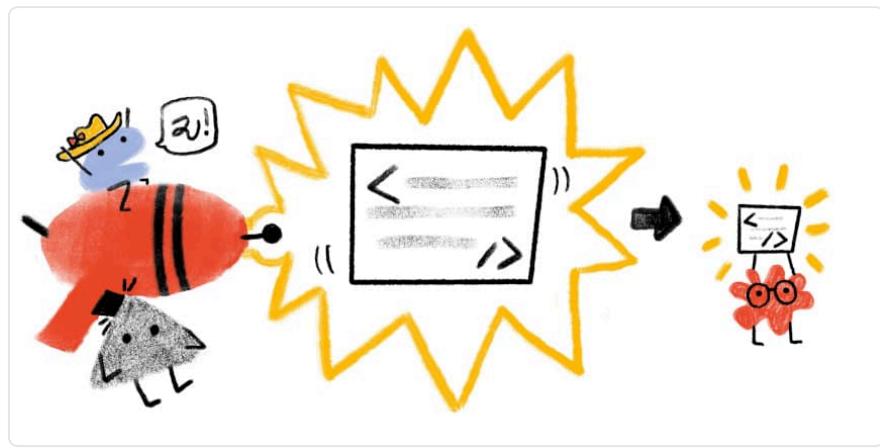
@HenriHelvetica henrihelvetica

Henri is a freelance developer who has turned his interests to a potpourri of performance engineering with pinches of user experience. When not reading the deluge of daily research docs and case studies, or indiscriminately auditing sites in devtools, Henri can be found contributing back to the community, co-programming meetups including the Toronto Web Performance Group or volunteering his time for lunch and learns at various bootcamps. Otherwise, he's tooling with music production software or with near certainty training and focusing on running the fastest 5k possible.



Part IV Chapter 19

Compression [UNEDITED]



Written by Moritz Firsching, Luca Versari, Sami Boukortt, and Jyrki Alakuijala

Reviewed by Paul Calvano

Analyzed by Abby Tsai

Introduction

Using HTTP compression makes a website load faster and therefore guarantees a better user experience. Running no compression on HTTP makes for a worse user experience, may affect the growth rate of the related web service, and affects search rankings. Effective use of compression can reduce page weight, improves web performance, and therefore is an important part of search engine optimization.

While lossy compression is often acceptable for images and other media types, for text we want to use lossless compression, i.e. recover the exact text after decompression.

What type of content should we compress?

For most text-based assets, such as HTML, CSS, JavaScript, JSON, or SVG, as well as certain

non-text formats such as woff, ttf, ico, using compression is recommended.

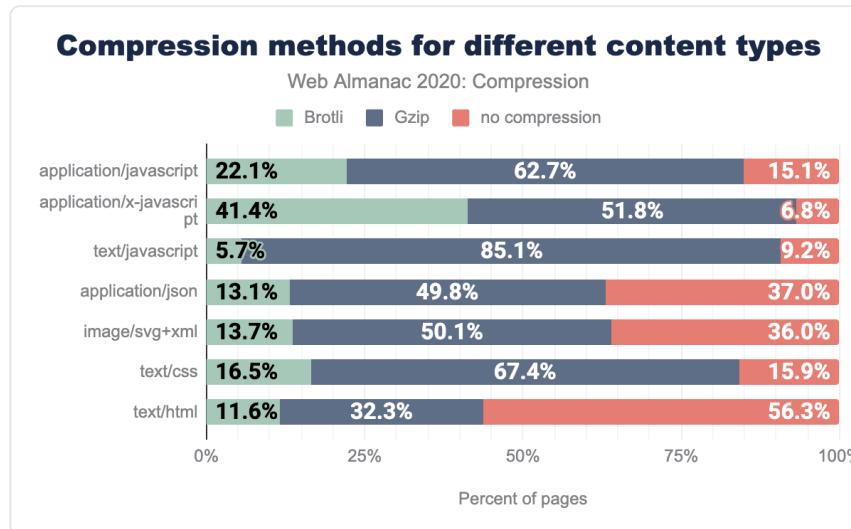


Figure 19.1. Compression Methods for Different Content Types

The figure shows the percent of requests of a certain content type using either Brotli, Gzip or no text compression. It is surprising that while all those content types would profit from compression, the range of percentages varies widely over the different content types: only 44% use compression for `text/html` against 93% for `application/x-javascript`.

For image-based assets text-based compression is less useful and not widely employed. The data shows that the percent of image requests that employ either Brotli, or Gzip is very low, less than 4%. For more info on non text-based assets, check out the Media chapter.

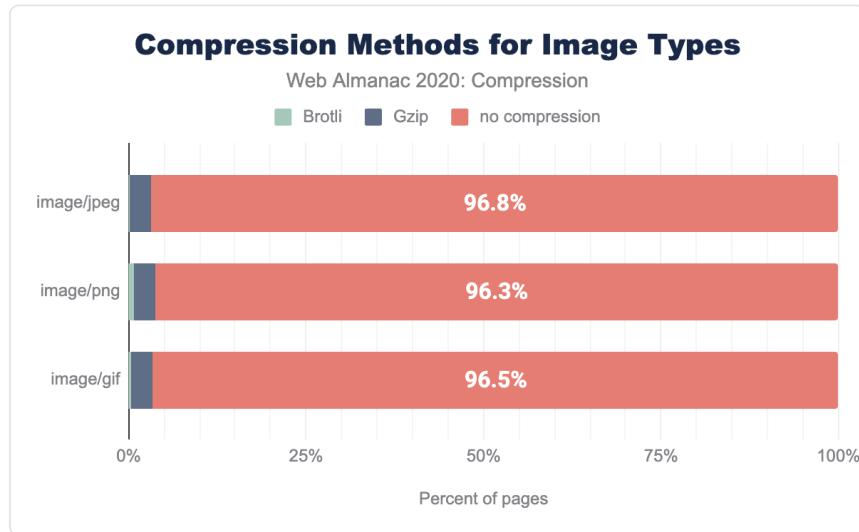


Figure 19.2. Compression by content type as a percent for desktop.

How to use HTTP compression?

To reduce the size of the files that we plan to serve one could first use some minimizers, e.g. HTMLMinifier, CSSNano, or UglifyJS. However bigger gains are expected from using compression.

There are two ways of doing the compression on the server side:

- Precompressed (compress and save assets ahead of time)
- Dynamically Compressed (compress assets on-the-fly after a request is made)

Since precompression is done beforehand, we can spend more time compressing the assets. For dynamically compressed resources, we need to choose the compression levels such that compression takes less time than the time difference between sending an uncompressed versus a compressed file. This difference is borne out when looking at compression level recommendations for both methods.

	Brotli	Gzip
precompressed	11	9 or Zopfli
dynamically compressed	5	6

Figure 19.3. Recommended compression levels to use.

Currently, practically all text compression is done by one of two HTTP content encodings: Gzip and Brotli. Both are widely supported by browsers: can I use Brotli?/can I use Gzip

When you want to use Gzip, consider using Zopfli, which generates smaller Gzip compatible files. This should be done especially for precompressed resources, since here the greatest gains are expected. See this comparison between Gzip and Zopfli that takes into account different compression levels for Gzip.

Many popular servers support dynamically and/or pre-compressed HTTP and many of them support Brotli.

Current state of HTTP compression

Approximately 60% of HTTP responses are delivered with no text-based compression. This may seem like a surprising statistic, but keep in mind that it is based on all HTTP requests in the dataset. Some content, such as images, will not benefit from these compression algorithms and is therefore not often used, as shown in figure 19.2.

Content Encoding	Desktop	Mobile	Combined
No Text Compression	60.06%	59.31%	59.67%
Gzip	30.82%	31.56%	31.21%
<i>br</i>	9.10%	9.11%	9.11%
Other	0.02%	0.02%	0.02%

Figure 19.4. Adoption of compression algorithms.

Of the resources that are served compressed, the majority are using either Gzip (77%) or Brotli (23%). The other compression algorithms are used infrequently.

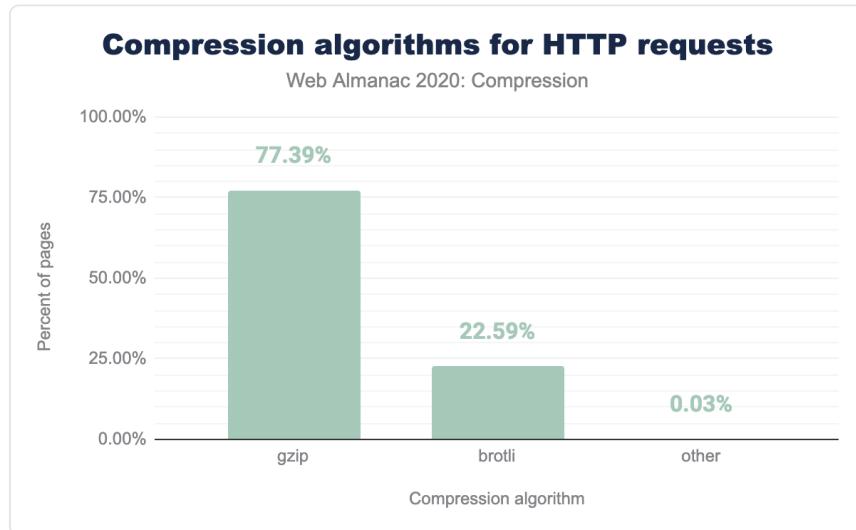


Figure 19.5. Compression algorithm usage rates.

In the graph below, the top 11 content types are displayed with box sizes representing the relative number of requests. The color of each box represents how many of these resources were served compressed, orange indicates a low percentage of compression while blue indicates a high percentage of compression. Most of the media content is shaded orange, which is expected since Gzip and Brotli would have little to no benefit for them. Most of the text content is shaded blue to indicate that they are being compressed. However, the light blue shading for some content types indicate that they are not compressed as consistently as the others.

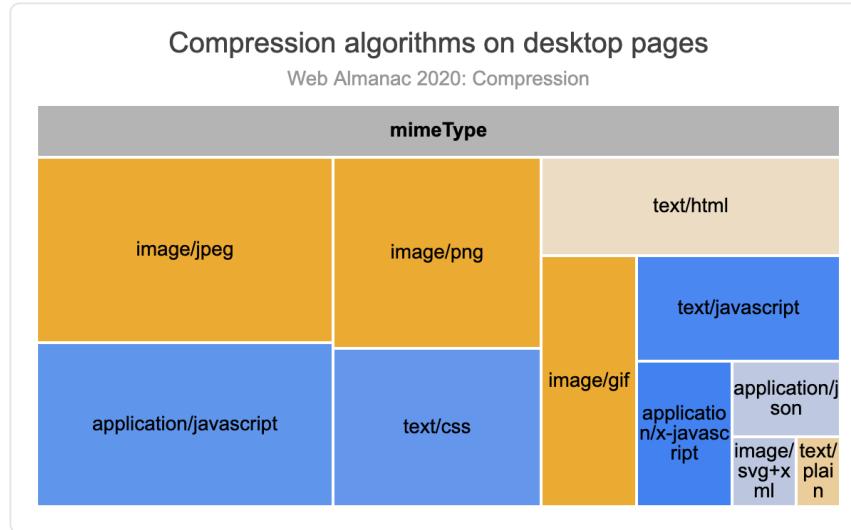


Figure 19.6. Top compressed content types on desktop.

Figure 19.1 above breaks down the percentage of compression used per content type, in figure 19.6 this percentage is indicated as color. The two figures tell similar stories, non-text based assets are rarely compressed, while text-based assets are often compressed. The rates of compression are also similar for both mobile and desktop.

First-party vs third-party compression

In the Third Parties chapter, we learn about third parties and their impact on performance. This is also true for third-party request compression.

	Desktop	Mobile		
Content Encoding	First-Party	Third-Party	First-Party	Third-Party
No Text Compression	61.93%	57.81%	60.36%	58.11%
Gzip	30.95%	30.66%	32.36%	30.65%
br	7.09%	11.51%	7.26%	11.22%
deflate	0.02%	0.01%	0.02%	0.01%
Other / Invalid	0.01%	0.01%	0.01%	0.01%

Figure 19.7. First-party versus third-party compression by device type.

When we compare compression techniques between first and third parties, we can see that third-party content tends to be compressed more than first-party content. Additionally, the percentage of Brotli compression is higher for third-party content. This is likely due to the number of resources served from the larger third parties that typically support Brotli, such as Google and Facebook.

Compared with last year's results, we can see that there was a significant increase in the use of compression, notably Brotli for first parties, almost to the point that the use of compression is around 40% for both first and third party, and for desktop and mobile. However within the requests that do use compression, for first parties, the ratio of Brotli compression is only 18%, while the ratio for third parties is 27%.

How to analyze compression on your sites

You can use Firefox Developer Tools or Chrome DevTools to quickly figure out what content a website already compresses. To do this, go to the Network tab, right click and activate "Content Encoding" under Response Headers. Hovering over the size of individual files you will see "transferred over network" and "resource size". Aggregated for the entire site one can see size/transferred size for Firefox and "transferred" and "resources" for Chrome on the bottom left hand side of the Network tab.

The screenshot shows the Network tab in Google Chrome DevTools. On the left, a list of network requests is displayed, including files like 'compression', 'normalize.css', and various CSS and JS files. On the right, a detailed view of a selected request's response headers is shown. A dropdown menu is open, listing various header fields. The 'Content-Encoding' field is highlighted with a yellow oval. At the bottom of the DevTools interface, a summary bar shows '31 requests | 654 kB transferred | 2.1 MB resources | Finish: 2.52 s | DOMContentLoaded: 1.41 s'.

Figure 19.8. Use DevTools to check if content encoding is used on your site

Another tool to better understand compression on your site is Google's Lighthouse tool, which enables you to run a series of audits against web pages. The text compression audit evaluates

whether a site can benefit from additional text-based compression. It does this by attempting to compress resources and evaluate whether an object's size can be reduced by at least 10% and 1,400 bytes. Depending on the score, you may see a compression recommendation in the results, with a list of specific resources that could be compressed.

Because the HTTP Archive runs Lighthouse audits for each mobile page, we can aggregate the scores across all sites to learn how much opportunity there is to compress more content. Overall, 74% of websites are passing this audit, while almost 13% of websites have scored below a 40. This is a 11.5% improvement when compared to last year's 62.5% of passing scores.

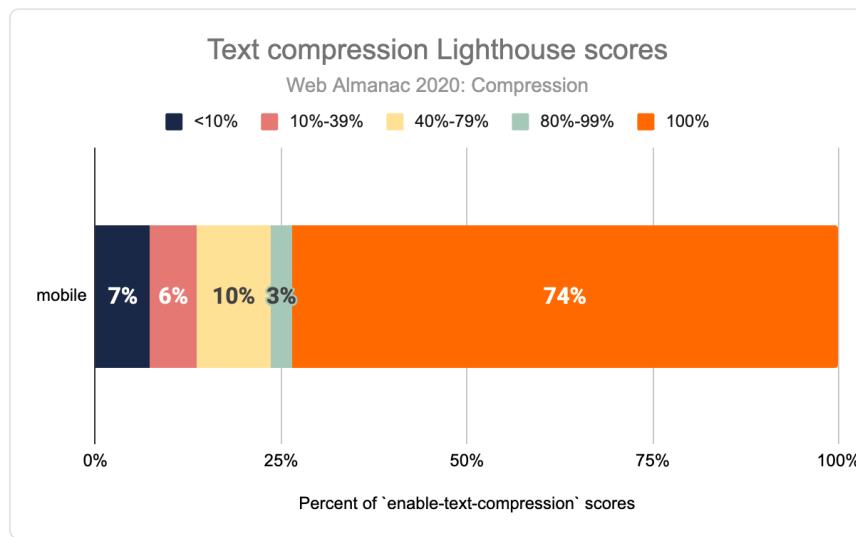


Figure 19.9. Lighthouse "enable text compression" audit scores.

Conclusion

Compared with last year's Almanac, there is a clear trend towards using more text compression. The number of requests that don't use any text compression went down a little more than 2%, while at the same time the use of Brotli has increased by almost 2%. The Lighthouse scores have improved significantly.

Text compression is widely used for the relevant formats, although there is still a significant percentage of HTTP requests that could benefit from additional compression. You can profit from taking a close look at the configuration of your server and set compression methods and levels to your need. A great impact for a more positive user experience could be made by carefully choosing defaults for the most popular HTTP servers.

Authors



Moritz Firsching

⌚ mo271 ⌐ <https://mo271.github.io/>

Moritz Firsching is software engineer at Google Switzerland, where he works on progressive image formats and font compression. Before that Moritz did research as a mathematician studying polytopes.



Luca Versari

⌚ veluca93

Luca Versari is a software engineer at Google, working on JPEG XL⁶⁰. He's finishing a PhD on graph compression and has a background in mathematics.



Sami Boukortt

⌚ sboukortt

Sami joined Google after completing his studies in engineering mathematics. After a few years of remote interest in compression, he eventually made it his full-time subject of work in 2018.



Jyrki Alakuijala

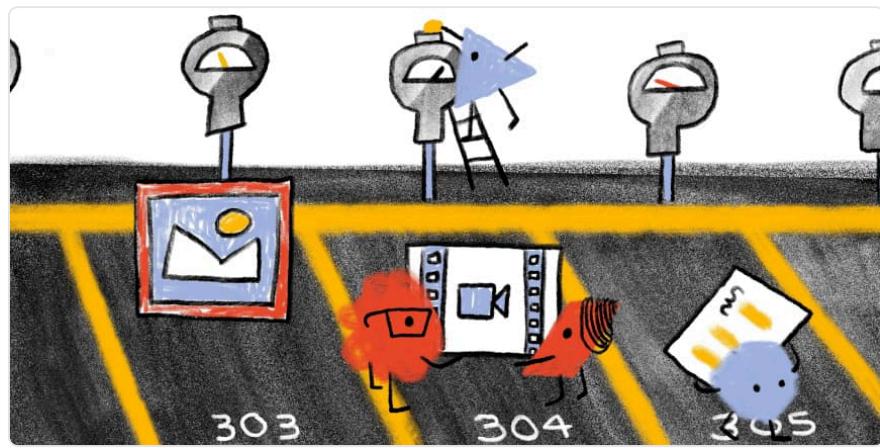
⌚ jyrkialakuijala

Jyrki Alakuijala is an active member of the open source software community, and a data compression researcher. Jyrki works at Google as a Technical Lead/Manager, and his recent published work has been with Zopfli, Butteraugli, Guetzli, Gipfeli, WebP lossless, Brotli, and JPEG XL compression formats and algorithms, and two hashing algorithms, CityHash, and HighwayHash. Before his Google employment he developed software for neurosurgery and radiation therapy treatment planning.

60. <https://gitlab.com/wg1/jpeg-xl>

Part IV Chapter 20

Caching [UNEDITED]



Written by **Rory Hewitt and Raghu Ramakrishnan**

Reviewed by **Julia Yang**

Analyzed by **Raghu Ramakrishnan**

Introduction

Caching is a technique that enables the reuse of previously downloaded content. It involves something (a server which builds web pages, a proxy such as a CDN or the browser itself) storing 'content' (web pages, CSS, JS, images, fonts, etc.) and tagging it appropriately, so it can be reused.

Here's a very high-level example:

Jane visits the home page of the www.example.com website. Jane lives in Los Angeles, CA, and the example.com server is located in Boston, MA. Jane visiting www.example.com involves a network request which has to travel across the country.

On the example.com server (a.k.a. Origin server), the home page is retrieved. The server knows Jane is located in LA and adds dynamic content to the page - a list of upcoming events near her. Then the page is sent back across the country to Jane and displayed on her browser.

If there is no caching, if Carlos in LA also visits www.example.com after Jane, his request must travel across the country to the example.com server. The server has to build the same page, including the LA events list. It will have to send the page back to Carlos.

Worse, if Jane revisits the example.com home page, her subsequent requests will act like the first - the request must go across the country and the example.com server must rebuild the home page to send it back to her.

So without any caching, the example.com server builds each request from scratch. That's bad for the server because it's more work. Additionally, any communication between either Jane or Carlos and the example.com server requires data to travel across the country. All of this can add up to a slow experience that's bad for both of them.

However, with server caching, when Jane makes her first request the server builds the LA variant of the home page. It caches the data for reuse by all LA visitors. So when Carlos's request gets to the example.com server, the server checks if it has the LA variant of the home page in its cache. Since that page is in cache as a result of Jane's earlier request, the server saves time by returning the cached page.

More importantly, with browser caching, when Jane's browser receives the page from the server for the first request, it caches the page. All of her future requests for the example.com home page will be served instantly from her browser's cache, without a network request. The example.com server also benefits by not having to process or deal with Jane's request.

Jane is happy. Carlos is happy. The example.com folks are happy. Everyone is happy.

It should be clear then, that browser caching provides a significant performance benefit by avoiding costly network requests. It also helps an application scale by reducing the traffic to a website's origin infrastructure. Server caching also significantly reduces the load on the underlying application.

Caching benefits both the end users (they get their web pages quickly) and the companies serving the web pages (reducing the load on their servers). Caching really is a win-win!

Web architectures typically involve multiple tiers of caching. There are four main places ('caching entities') where caching can occur:

1. An end user's web browser.
2. A service worker cache running in the end user's web browser.
3. A Content Delivery Network (CDN) or similar proxy, which sits between the end user's web browser and the origin server.
4. The origin server itself.

In this chapter, we will primarily be discussing caching within web browsers (1-2), as opposed to caching at the origin server or in a CDN. Nevertheless, many of the specific caching topics

discussed in this chapter rely on the relationship between the browser and the server (or CDN, if one is used).

The key to understanding how caching (and the web) works is to remember that it all consists of transactions between a requesting entity (e.g. a browser) and a responding entity (e.g. a server). Each transaction consists of two parts:

1. The request from the requesting entity ("I want object X"), and
2. The response from the responding entity ("Here is object X").

When we talk about caching, it refers to the object (HTML page, image, etc.) cached by the requesting entity.

Below figure shows how a typical request/response flow works for an object (e.g. a web page). A CDN sits between the browser and the server. Note that at each point in the browser → CDN → server flow, each of the caching entities first checks whether it has the object in its cache. It returns the cached object to the requester if found, before forwarding the request to the next caching entity in the chain:

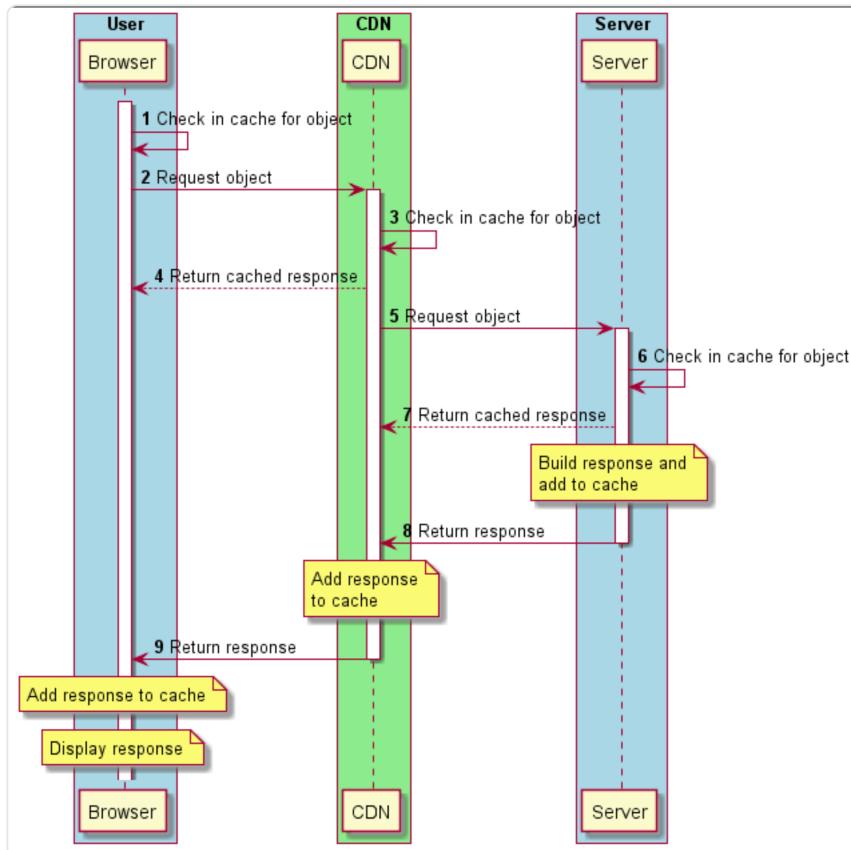


Figure 20.1. Request/response flow for an object.

Unless specified otherwise, all statistics in this chapter are for desktop, on the understanding that mobile statistics are similar. Where mobile and desktop statistics differ significantly, that is called out. Many of the responses used in this chapter are from web servers which use commonly-available server packages. While we may indicate 'best practices', the practices may not be possible if the software package used has a limited number of cache options.

Caching guiding principles

There are three guiding principles to caching web content:

- Cache as much as you can
- Cache for as long as you can

- Cache as close as you can to end users

Cache as much as you can

When considering what to cache, it is important to understand whether the response content is *static* or *dynamic*.

- An example of static content is an image. For instance, a picture of a cat is the same regardless of who's requesting it or where the requester is located.
- An example of dynamic content is a list of events which are specific to a geographic location. The list will be different based on the requester's location.



Figure 20.2. Yes, we have a picture of a cat.

Static content is typically cacheable and often for long periods of time. It has a one-to-many relationship between the content (one) and the requests (many).

Dynamically generated content can be more nuanced and requires careful consideration. Some dynamic content can be cached, but often for a shorter period of time. The example of a list of upcoming events will change, possibly from day to day. Different variants of the list may also need to be cached and what's cached in a user's browser may be a subset of what's cached on the server or CDN. Nevertheless, it is possible to cache some dynamic contents. It is incorrect

to assume that "dynamic" is another word for "uncacheable".

Cache for as long as you can

The length of time you would cache a resource is highly dependent on the content's *vulnerability* (the likelihood and/or frequency of change). For example, an image or a versioned JavaScript file could be cached for a very long time. An API response or a non-versioned JavaScript file may need a shorter cache duration to ensure users get the most up-to-date response. Some content might only be cached for a minute or less. And, of course, some content should not be cached at all. This is discussed in more detail in Identifying caching opportunities.

Another point to bear in mind is that no matter how long you tell a browser to cache content for, the browser may evict that content from cache before that point in time. It may do so to make room for other content that is accessed more frequently, etc.. However, a browser will never cache content for longer than it is told.

Cache as close to end users as you can

Caching content close to the end user reduces download times by removing latency. For example, if a resource is cached in a user's browser, then the request never goes out to the network and it is available instantaneously every time the user needs it. For visitors that don't have entries in their browser's cache, a CDN would be the next place a cached resource is returned from. In most cases, it will be faster to fetch a resource from a local cache or a CDN compared to an origin server.

Some terminology

- Caching entity - the hardware or software that is doing the caching. Due to the focus of this chapter, we use "browser" as a synonym for "caching entity" unless otherwise specified.
- TTL - the Time-To-Live of a cached object defines how long it can be stored in a cache, typically measured in seconds. After a cached object reaches its TTL, it is marked as 'stale' by the cache. Depending on how it was added to the cache (see the details of the caching headers below), it may be evicted from cache immediately, or it may remain in the cache but marked as a 'stale' object, requiring revalidation before reuse.
- Eviction - the automated process by which an object is actually removed from a cache when/after it reaches its TTL or possibly when the cache is full.

- Revalidation - a cached object that is marked as stale may need to be 'revalidated' with the server before it can be displayed to the user. The browser must first check with the server that the object the browser has in its cache is still up-to-date and valid.

Overview of browser caching

When a browser makes a request for a piece of content (e.g. a web page), it will receive a response which includes not just the content itself (the HTML markup), but also a number of response headers which describe the content, including information about its cacheability.

The caching-related headers, or the absence of them, tell the browser three important pieces of information:

- **Cacheability:** Is this content cacheable?
- **Freshness:** If it is cacheable, how long can it be cached for?
- **Validation:** If it is cacheable, how do I subsequently ensure that my cached version is still fresh?

The full specifications for these caching headers are in RFC 7234, and discussed in sections 4.2 (Freshness) and 4.3 (Validation).

The two HTTP response headers typically used for specifying freshness are `Cache-Control` and `Expires`:

- `Expires` specifies an explicit expiration date and time (i.e. when exactly the content expires)
- `Cache-Control` specifies a cache duration (i.e. how long the content can be cached in the browser relative to when it was requested)

Often, both these headers are specified; in that case `Cache-Control` takes precedence. These headers are discussed in more detail below.

Cache-Control vs Expires

In the early HTTP/1.0 days of the web, the `Expires` header was the only cache-related response header. As stated above, it is used to indicate the exact date/time after which the response is considered stale. Its value is a date and time, such as:

```
Expires: Thu, 01 Dec 1994 16:00:00 GMT
```

The `Expires` header can be thought of as a 'blunt instrument'. If a relative cache TTL is required, then processing must be done on the server to generate an appropriate value based upon the current date/time.

HTTP/1.1 introduced the `Cache-Control` header, which is supported by all modern browsers. The `Cache-Control` header provides much more extensibility and flexibility than `Expires` via *caching directives*, several of which can be specified together. Details on the various directives are below.

The simple example below shows a request and response for a JavaScript file (some headers have been removed for clarity). The `Date` header indicates the current date (specifically, the date that the content was served). The `Expires` header indicates that it can be cached for 10 minutes (the difference between the `Expires` and `Date` headers). The `Cache-Control` header specifies the `max-age` directive, which indicates that the resource can be cached for 600 seconds (5 minutes). Since `Cache-Control` takes precedence over `Expires`, the browser will cache the response for 5 minutes, after which it will be marked as stale:

```
> GET /static/js/main.js HTTP/2
> Host: www.example.org
> Accept: */*
< HTTP/2 200
< Date: Thu, 23 Jul 2020 03:04:17 GMT
< Expires: Thu, 23 Jul 2020 03:14:17 GMT
< Cache-Control: public, max-age=600
```

RFC 7234 says that if no caching headers are present in a response, then the browser is allowed to *heuristically* cache the response - it suggests a cache duration of 10% of the time since the `Last-Modified` header (if passed). In such cases, most browsers implement a variation of this suggestion, but some may cache the response indefinitely and some may not cache it at all. Because of this variation between browsers, it is important to explicitly set specific caching rules to ensure that you are in control of the cacheability of your content.

- 73.6% of responses are served with a `Cache-Control` header
- 55.5% of responses are served with an `Expires` header
- 54.8% of responses include both headers
- 25.6% of responses did not include either header, and are therefore subject to heuristic caching

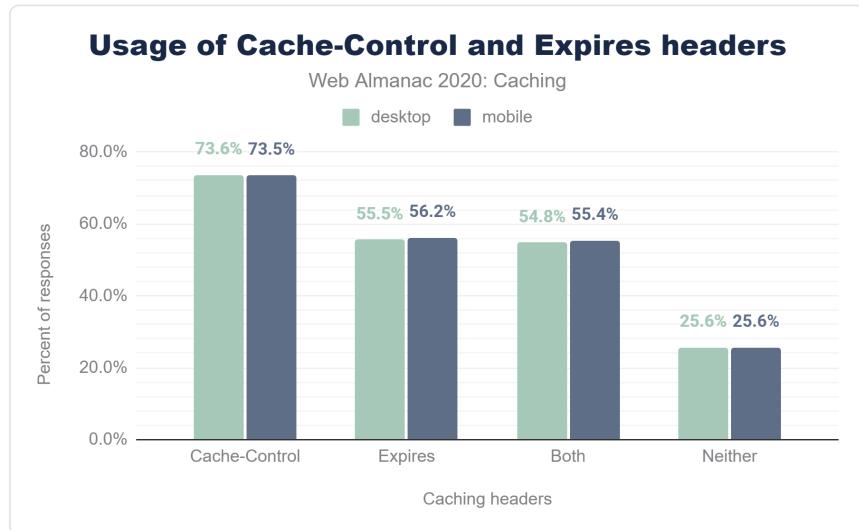


Figure 20.3. Usage of `Cache-Control` and `Expires` headers.

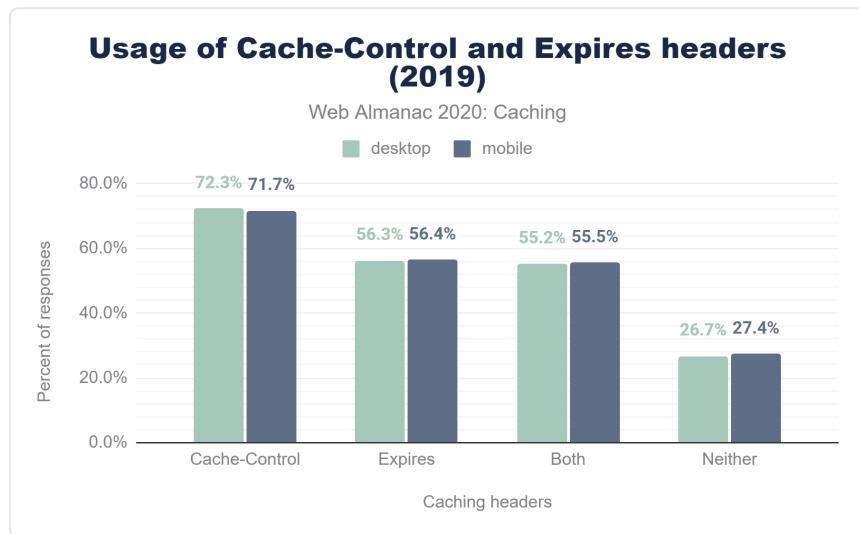


Figure 20.4. Usage of `Cache-Control` and `Expires` headers in 2019.

These statistics are interesting since, compared with 2019, while we see an increase in the use of the `Cache-Control` header (1.3%), we also see a minimal decrease in the use of the older `Expires` header (0.7%). Effectively, a percentage of servers are merely adding the `Cache-Control` header to their responses without removing the `Expires` header.

As we delve into the various directives allowed in the `Cache-Control` header, we will see how its flexibility and power make it a better fit in many cases.

Cache-Control directives

When you use the `Cache-Control` header, you specify one or more *directives* - predefined values that indicate specific caching functionality. Multiple directives are separated by commas and can be specified in any order, although some of them 'clash' with one another (e.g. `public` and `private`). Some directives take a value, such as `max-age`.

Below is a table showing the most common `Cache-Control` directives:

Directive	Description
<code>max-age</code>	Indicates the number of seconds that a resource can be cached for, relative to the current time. For example ' <code>max-age=86400</code> '.
<code>public</code>	Any cache may store the response, including the browser, and any proxies between the server and the browser, such as a CDN. This is assumed by default.
<code>no-cache</code>	A cached entry must be revalidated prior to its use, via a conditional request, even if it is not marked as stale.
<code>must-revalidate</code>	A stale cached entry must be revalidated prior to its use, via a conditional request.
<code>no-store</code>	Indicates that the response must not be cached.
<code>private</code>	The response is intended for a specific user and should not be stored by shared caches such as proxies and CDNs.
<code>proxy-revalidate</code>	Same as <code>must-revalidate</code> but applies to shared caches.
<code>s-maxage</code>	Same as <code>max-age</code> but applies to shared caches (e.g. CDN's) only.
<code>immutable</code>	Indicates that the cached entry will never change during its TTL, and that revalidation is not necessary.
<code>stale-while-revalidate</code>	Indicates that the client is willing to accept a stale response while asynchronously checking in the background for a fresh one.
<code>stale-if-error</code>	Indicates that the client is willing to accept a stale response if the check for a fresh one fails.

Figure 20.5. `Cache-Control` directives.

The `max-age` directive is the most commonly-found, since it directly defines the TTL, in the

same way that the `Expires` header does.

Here is an example of a valid Cache-Control header with multiple directives:

```
Cache-Control: public, max-age=86400, must-revalidate
```

This indicates that the object can be cached for 86,400 seconds (1 day) and it can be stored by all caches between the server and the browser, as well as in the browser itself. Once it has reached its TTL and is marked as stale, it can remain in cache, but must be conditionally revalidated before reuse.

- 60.2% of responses include a `Cache-Control` header with the `max-age` directive.
- 45.5% of responses include the `Cache-Control` header with the `max-age` directive and the `Expires` header, which means that 10% of responses are caching solely based on the older `Expires` header.

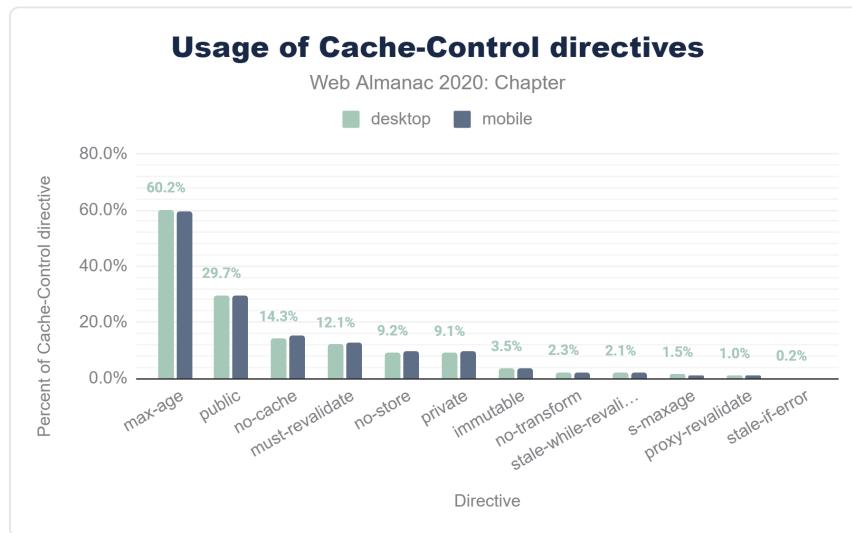


Figure 20.6. Distribution of `Cache-Control` directives.

The above figure illustrates the 11 `Cache-Control` directives in use on mobile and desktop websites. There are a few interesting observations about the popularity of these cache directives:

- `max-age` is used by about 60.2% of `Cache-Control` headers, and `no-store` is used by about 9.2% (see below for some discussion on the meaning and use of the `no-store` directive).

- Explicitly specifying `public` isn't ever really necessary since cached entries are assumed `public` unless `private` is specified. Nevertheless, almost one third of responses include `public` - a waste of a few header bytes on every response :)
- The `immutable` directive is relatively new, introduced in 2017 and is only supported on Firefox and Safari - its usage is still only at about 3.5%, but it is widely seen in responses from Facebook, Google, Wix, Shopify and others. It has the potential to greatly improve cacheability for certain types of requests.

As we head out to the long tail, there are a small percentage of 'invalid' directives that can be found; these are ignored by browsers, and just end up wasting header bytes. Broadly they fall into two categories:

- Misspelled directives such as `nocache` and `s-max-age` and invalid directive syntax, such as using `:` instead of `=` or using `_` instead of `-`.
- Non-existent directives such as `max-stale`, `proxy-public`, `surrogate-control`.

The most interesting standout in the list of invalid directives is the use of `no-cache="set-cookie"` (even at only 0.2% of all `Cache-Control` header values, it still makes up more than all the other invalid directives combined). In some early discussions on the `Cache-Control` header, this syntax was raised as a possible way to ensure that any `Set-Cookie` response headers (which might be user-specific) would not be cached with the object itself by any intermediate proxies such as CDNs. However, this syntax was not included in the final RFC; nearly equivalent functionality can be implemented using the `private` directive, and the `no-cache` directive does not allow a value.

Cache-Control: no-store, no-cache and max-age=0

When a response absolutely must not be cached, the `Cache-Control no-store` directive should be used; if this directive is not specified, then the response is considered *cacheable and may be cached*. Note that if `no-store` is specified, it takes precedence over other directive - this makes sense, since serious privacy and security issues could occur if a resource is cached which should not be.

We can see a few common errors that are made when attempting to configure a response to be non-cacheable:

- Specifying `Cache-Control: no-cache` may sound like a directive to not cache the resource. However, as noted above, the `no-cache` directive does allow the

resource to be cached - it simply informs the browser to revalidate the resource prior to use and is not the same as stopping the resource from being cached at all.

- Setting `Cache-Control: max-age=0` sets the TTL to 0 seconds, but again, that is not the same as being `non-cacheable`. When `max-age=0` is specified, the resource is cached, but is marked as stale, resulting in the browser having to immediately revalidate its freshness.

Functionally, `no-cache` and `max-age=0` are similar, since they both require revalidation of a cached resource. The no-cache directive can also be used alongside a `max-age` directive that is greater than 0 - this results in the object being cached for the specified TTL, but being revalidated prior to every use.

When looking at the above three discussed directives, 2.3% of responses include the combination of all three `no-store`, `no-cache` and `max-age=0` directives, 6.6% of responses include both `no-store` and `no-cache`, and a negligible number of responses (< 1%) include `no-store` alone.

As noted above, where `no-store` is specified with either/both of `no-cache` and `max-age=0`, the `no-store` directive takes precedence, and the other directives are ignored. Therefore, if you don't want content to be cached anywhere, simply specifying `Cache-Control: no-store` is sufficient, and is both simple and uses the minimum number of header bytes.

The `max-age=0` directive is present on less than 2% of responses where `no-store` is not specified. In such cases, the resource will be cached in the browser but will require revalidation as it is immediately marked as stale.

Conditional requests and revalidation

There are often cases where a browser has previously requested an object and already has it in its cache but the cache entry has already exceeded its TTL (and is therefore marked as stale) or where the object is defined as one that must be revalidated prior to use.

In these cases, the browser can make a conditional request to the server - effectively saying "*I have object X in my cache - can I use it, or do you have a more recent version I should use instead?*".

The server can respond in one of two ways:

- "*Yes, the version of object X you have in cache is fine to use*" - in this case the server response consists of a `304 Not Modified` status code and response headers, but no response body
- "*No, here is a more recent version of object X - use this instead*" - in this case the server

response consists of a `200 OK` status code, response headers, and a new response body (the actual new version of object X)

In either case, the server can optionally include updated caching response headers, possibly extending the TTL of the object so the browser can use the object for a further period of time without needing to make more conditional requests.

The above is known as *revalidation* and if implemented correctly can significantly improve perceived performance - since a `304 Not Modified` response consists only of headers, it is much smaller than a `200 OK` response, resulting in reduced bandwidth and a quicker response.

So how does the server identify a conditional request from a regular request?

It actually all comes down to the initial request for the object. When a browser requests an object which it does not already have in its cache, it simply makes a GET request, like this (some headers removed for clarity):

```
> GET /index.html HTTP/2
> Host: www.example.org
> Accept: */*
```

If the server wants to allow the browser to make use of conditional requests (this decision is entirely up to the server!), it can include one or both of two response headers which identify the object as being eligible for subsequent conditional requests. The two response headers are:

- `Last-Modified` - this indicates when the object was last changed. Its value is a date timestamp.
- `ETag` (Entity Tag) - this provides a unique identifier for the content as a quoted string. It can take any format that the server chooses; it is typically a hash of the file contents, but it could be a timestamp or a simple string.

If both headers are present, `ETag` takes precedence.

Last-Modified

When the server receives the request for the file, it can include the date/time that the file was most recently changed as a response header, like this:

```
< HTTP/2 200
< Date: Thu, 23 Jul 2020 03:04:17 GMT
< Last-Modified: Mon, 20 Jul 2020 11:43:22 GMT
< Cache-Control: max-age=600

...lots of html here...
```

The browser will cache this object for 600 seconds (as defined in the `Cache-Control` header), after which it will mark the object as stale. If the browser needs to use the file again, it requests the file from the server just as it did initially, but this time it includes an additional request header, called `If-Modified-Since`, which it sets to the value that was passed in the `Last-Modified` response header in the initial response:

```
> GET /index.html HTTP/2
> Host: www.example.org
> Accept: /*
> If-Modified-Since: Mon, 20 Jul 2020 11:43:22 GMT
```

When the server receives this request, it can check whether the object has changed by comparing the `If-Modified-Since` header value with the date that it most recently changed the file.

If the two values are the same, then the server knows that the browser has the latest version of the file and the server can return a `304 Not Modified` response with just headers (including the same `Last-Modified` header value) and no response body:

```
< HTTP/2 304
< Date: Thu, 23 Jul 2020 03:14:17 GMT
< Last-Modified: Mon, 20 Jul 2020 11:43:22 GMT
< Cache-Control: max-age=600
```

However, if the file on the server has changed since it was last requested by the browser, then the server returns a `200 OK` response consisting of headers (including an updated `Last-Modified` header) and the new version of the file in the body:

```
< HTTP/2 200
< Date: Thu, 23 Jul 2020 03:14:17 GMT
< Last-Modified: Thu, 23 Jul 2020 03:12:42 GMT
< Cache-Control: max-age=600

...lots of html here...
```

As you can see, the `Last-Modified` response header and `If-Modified-Since` request header work as a pair.

ETag

The functionality here is almost exactly the same as the date-based `Last-Modified` / `If-Modified-Since` conditional request processing described above.

However, in this case, the Server sends an `ETag` response header - rather than a date timestamp, an `ETag` is simply a string - often a hash of the file contents or a version number calculated by the server. The format of this string is entirely up to the server - the only important fact is that the server changes the `ETag` value whenever it changes the file.

In this example, when the server receives the initial request for the file, it can return the file's version in an `ETag` response header, like this:

```
< HTTP/2 200
< Date: Thu, 23 Jul 2020 03:04:17 GMT
< ETag: "v123.4.01"
< Cache-Control: max-age=600

...lots of html here...
```

As with the `If-Modified-Since` example above, the browser will cache this object for 600 seconds, as defined in the `Cache-Control` header. When it needs to request the object from the server again, it includes an additional request header, called `If-None-Match`, which has the value passed in the `ETag` response header in the initial response:

```
> GET /index.html HTTP/2
> Host: www.example.org
> Accept: /*
> If-None-Match: "v123.4.01"
```

When the server receives this request, it can check whether the object has changed by comparing the `If-None-Match` header value with the current version it has of the file. If the two values are the same, then the browser has the latest version of the file and the server can return a `304 Not Modified` response with just headers:

```
< HTTP/2 304
< Date: Thu, 23 Jul 2020 03:14:17 GMT
< ETag: "v123.4.01"
< Cache-Control: max-age=600
```

However, if the values are different, then the version of the file on the server is more recent than the version that the browser has, so the server returns a `200 OK` response consisting of headers (including an updated `ETag` header) and the new version of the file:

```
< HTTP/2 200
< Date: Thu, 23 Jul 2020 03:14:17 GMT
< ETag: "v123.5.06"
< Cache-Control: public, max-age=600
...lots of html here...
```

Again, we see a pair of headers being used for this conditional request processing - the `ETag` response header and the `If-None-Match` request header.

In the same way that the `Cache-Control` header has more power and flexibility than the `Expires` header, the `ETag` header is in many ways an improvement over the `Last-Modified` header. There are two reasons for this:

1. The server can define its own format for the `ETag` header. The example above shows a version string, but it could be a hash, or a random string. By allowing this, versions of an object are not explicitly linked to dates, and this allows a server to

create a new version of a file and yet give it the same ETag as the prior version - perhaps if the file change is unimportant

2. ETags can be defined as either 'strong' or 'weak', which allows browsers to validate them differently. A full understanding and discussion of this functionality is beyond the scope of this chapter, but can be found in RFC 7232.
- 73.5% of responses are served with a Last-Modified header. Its usage has marginally increased (by < 1%) in comparison to 2019.
 - 47.9% of responses are served with an ETag header. Out of these responses, 36% are 'strong', 98.2% are 'weak', and the remaining 1.8% are invalid. In contrast with Last-Modified, the usage of ETag headers has marginally decreased (by <1%) in comparison to 2019.
 - 42.8% of responses are served with both headers (as noted above, in this case, the ETag header takes precedence).
 - 21.4% of responses include neither a Last-Modified or ETag header.

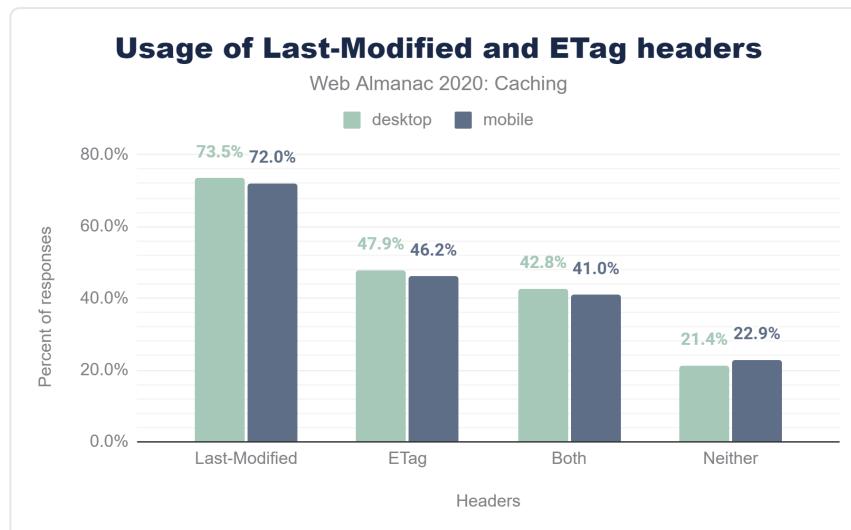


Figure 20.7. Adoption of validating freshness via Last-Modified and ETag headers.

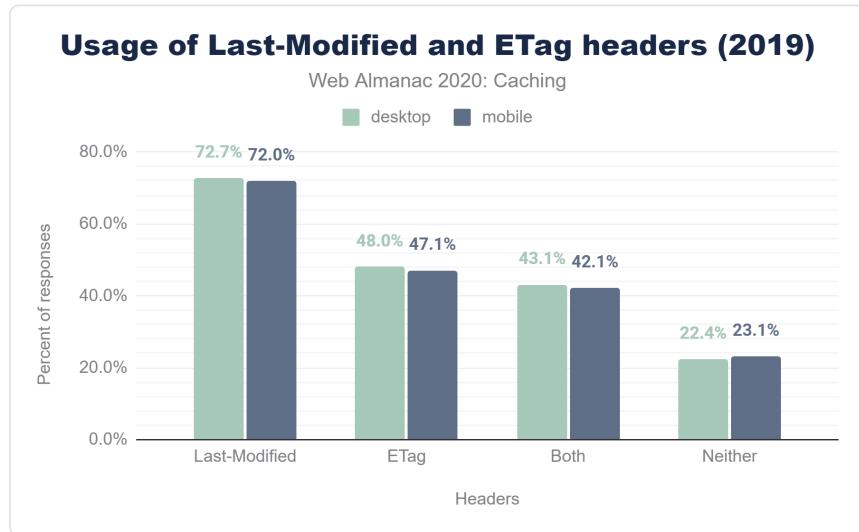


Figure 20.8. Adoption of validating freshness via `Last-Modified` and `ETag` headers in 2019.

Correctly-implemented revalidation using conditional requests can significantly reduce bandwidth (304 responses are typically much smaller than 200 responses), load on servers (only a small amount of processing is required to compare change dates or hashes) and improve perceived performance (servers respond more quickly with a 304). However, as we can see from the above statistics, more than a fifth of all requests are not using any form of conditional requests.

- Only 0.1% of the responses had a `304 Not Modified` status.
- 20.5% of the responses had no ETag header and contained the same `Last-Modified` value, passed in the `If-Modified-Since` header of the corresponding request. Out of these, 86% had a `304 Not Modified` status.
- 86.1% of the responses contained the same `ETag` value, passed in the `If-None-Match` header of the corresponding request. If the `If-Modified-Since` header is also present, `ETag` takes precedence. Out of these, 88.9% had a `304 Not Modified` status.

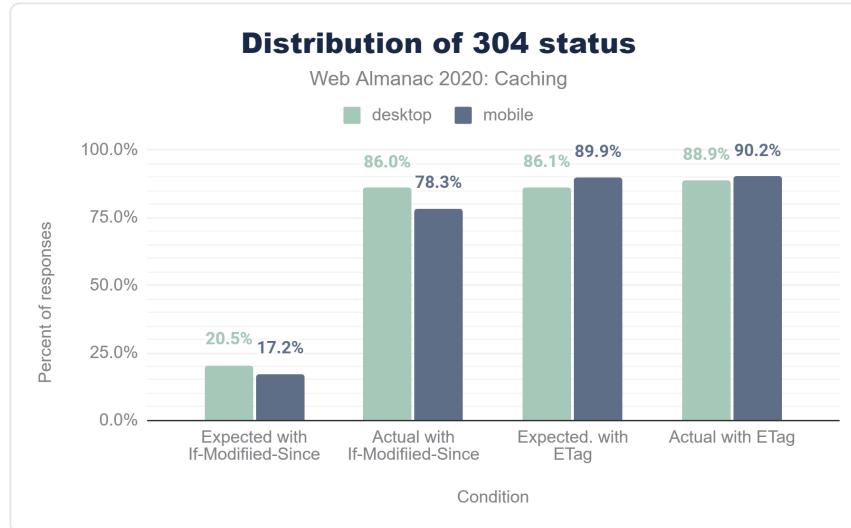


Figure 20.9. Distribution of `304 Not Modified` status.

Validity of date strings

Throughout this document, we have discussed several caching-related HTTP headers used to convey timestamps:

- The `Date` response header indicates when the resource was served to a client.
- The `Last-Modified` response header indicates when a resource was last changed on the server.
- The `Expires` header is used to indicate for how long a resource is cacheable.

All three of these HTTP headers use a date formatted string to represent timestamps. The date-formatted string is defined in RFC 2616, and must specify the 'GMT' timestamp string. For example:

```
> GET /index.html HTTP/2
> Host: www.example.org
> Accept: */*
```

```
< HTTP/2 200
```

```
< Date: Thu, 23 Jul 2020 03:14:17 GMT
< Cache-Control: max-age=600
< Last-Modified: Mon, 20 Jul 2020 11:43:22 GMT
```

Invalid date strings are ignored by most browsers, which can affect the cacheability of the response on which they are served - for example, an invalid `Last-Modified` header will result in the browser being unable to subsequently perform a conditional request for the object, since it is cached without that invalid timestamp.

Because the `Date` HTTP response header is almost always generated automatically by the web server, invalid values are extremely rare. Similarly `Last-Modified` headers had a very low percentage (0.5%) of invalid values. What was very surprising to see though, was that a relatively high 2.9% of `Expires` headers used an invalid date format (2.5% in mobile).

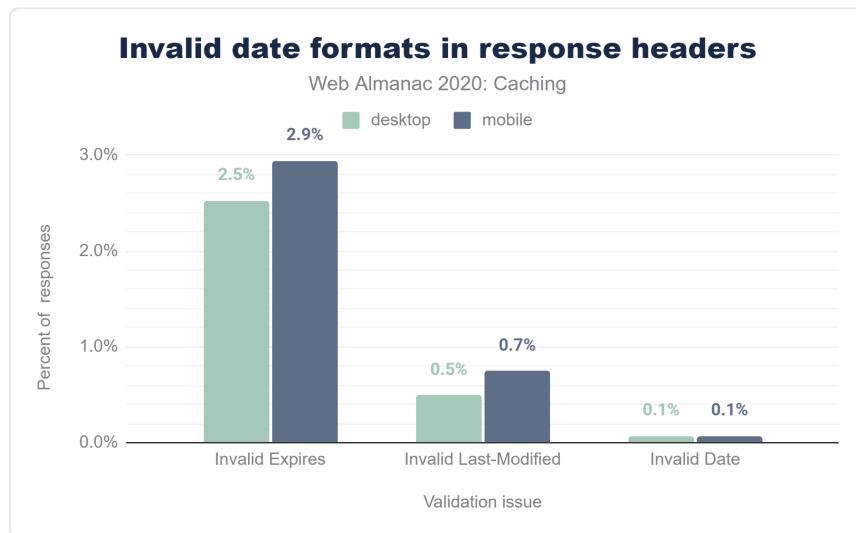


Figure 20.10. Invalid date formats in response headers.

Examples of some of the invalid uses of the `Expires` header are:

- Valid date formats, but using a time zone other than 'GMT'
- Numerical values such as 0 or -1
- Values that would be valid in a `Cache-Control` header

One large source of invalid `Expires` headers is from assets served from a popular third party, in which a date/time uses the EST time zone, for example `Expires: Tue, 27 Apr 1971`

19:44:06 EST . Note that some browsers may understand and accept this date format, on the principle of robustness, but it should not be assumed that this will be the case.

The `Vary` header

We have discussed how a caching entity can determine whether a response object is cacheable, and for how long it can be cached. However, one of the most important steps the caching entity must take is determining if the resource being requested is already in its cache. While this may seem simple, many times the URL alone is not enough to determine this. For example, requests with the same URL could vary in what compression they used (Gzip, Brotli, etc.) or could be returned in different encodings (XML, JSON etc.).

To solve this problem, when a caching entity caches an object, it gives the object a unique identifier (a cache key). When it needs to determine whether the object is in its cache, it checks for the existence of the object using the cache key as a lookup. By default, this cache key is simply the URL used to retrieve the object, but servers can tell the caching entity to include other 'attributes' of the response (such as compression method) in the cache key, by including the `Vary` response header, to ensure that the correct object is subsequently retrieved from cache - the `Vary` header identifies 'variants' of the object, based on factors other than the URL.

The `Vary` response header instructs the browser to add the value of one or more request header values to the cache key. The most common example of this is `Vary: Accept-Encoding`, which will result in the browser caching the same object in different formats, based on the different `Accept-Encoding` request header values (i.e. `gzip, br, deflate`).

A caching entity sends a request for an HTML file, indicating that it will accept a gzipped response:

```
> GET /index.html HTTP/2
> Host: www.example.org
> Accept-Encoding: gzip
```

The server responds with the object, and indicates that the version it is sending should include the value of the `Accept-Encoding` request header.

```
< HTTP/2 200 OK
```

```
< Content-Type: text/html  
< Vary: Accept-Encoding
```

In this simplified example, the caching entity would cache the object using a combination of the URL and the `Vary` header.

Another common value is `Vary: Accept-Encoding, User-Agent`, which instructs the client to include both the `Accept-Encoding` and `User-Agent` values in the cache key. When used from a browser, this might not make much sense - each browser has its own User-Agent value, so a browser would not make a request using different `User-Agent` values anyway. However, when discussing shared proxies and CDNs, using values other than `Accept-Encoding` can be problematic as it dilutes ('fragments') the cache and can reduce the amount of traffic served from cache. For instance, if a CDN attempts to cache many different variants of an object, including not just the URL and the `Accept-Encoding` header but also the `User-Agent` string (of which there are several thousand different varieties), it may end up filling up the cache with many almost identical (or indeed, identical) cached objects. This is very inefficient, and can lead to very sub-optimal caching within the CDN, resulting in fewer cache hits and greater latency. In general, you should only vary the cache if you are serving alternate content to clients based on that header.

The `Vary` header is used on 43.4% of HTTP responses, and 84.2% of these responses include a `Cache-Control` header.

The graph below details the popularity for the top 10 `Vary` header values. `Accept-Encoding` accounts for almost 92% of `Vary`'s use, with `User-Agent` at 10.7%, `Origin` (used for CORS processing) at 8%, and `Accept` at 4.1% making up much of the rest.

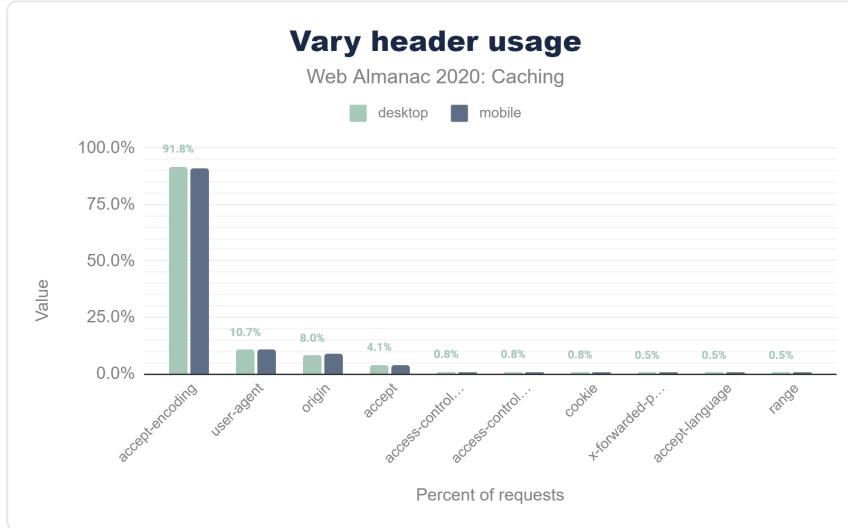


Figure 20.11. *Vary* header usage.

Setting cookies on cacheable responses

When a response is cached, its entire set of response headers are included with the cached object as well. This is why you can see the response headers when inspecting a cached response in Chrome via DevTools:

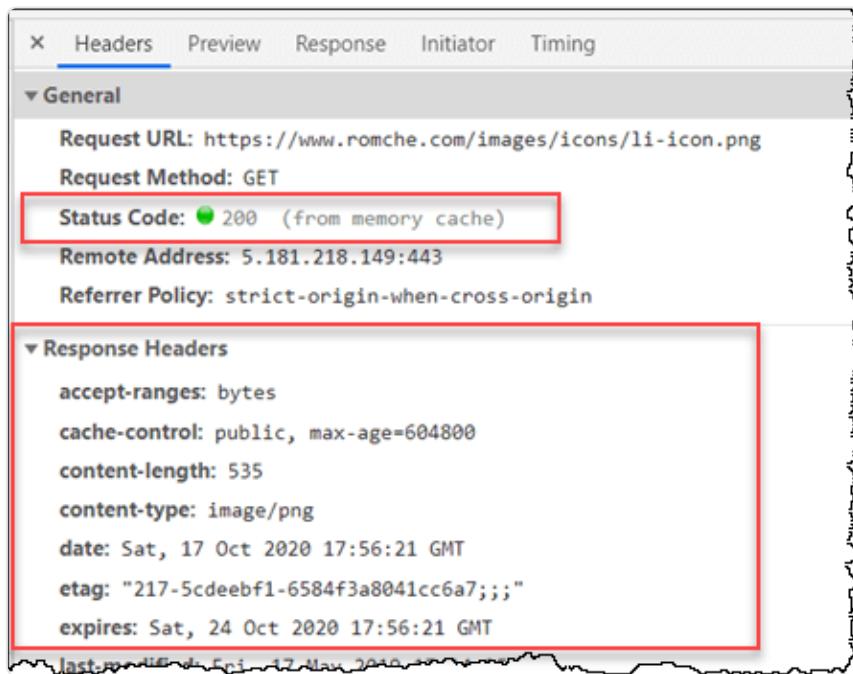


Figure 20.12. Chrome Dev Tools for a cached resource.

But what happens if you have a `Set-Cookie` on a response? According to RFC 7234 Section 8, the presence of a `Set-Cookie` response header does not inhibit caching. This means that a cached entry might contain a `Set-Cookie` response header. The RFC goes on to recommend that you should configure appropriate `Cache-Control` headers to control how responses are cached.

Since we have primarily been talking about browser caching, you may think this isn't a big issue - the `Set-Cookie` response headers that were sent by the server to me in responses to my requests clearly contain my cookies, so there's no problem if my browser caches them. However, if there is a CDN between myself and the server, the server must indicate to the CDN that the response should not be cached in the CDN itself, so that the response meant for me is not cached and then served (including my `Set-Cookie` headers!) to other users.

For example, if a login cookie or a session cookie is present in a CDN's cached object, then that cookie could potentially be reused by another client. The primary way to avoid this is for the server to send the `Cache-Control: private` directive, which tells the CDN not to cache the response, because it may only be cached by the client browser.

41.4% of cacheable responses contain a `Set-Cookie` header. Of those responses, only 4.6%

use the `private` directive. The remaining 95.4% (189.2 million HTTP responses) contain at least one `Set-Cookie` response header and can be cached by both public cache servers, such as CDNs. This is concerning and may indicate a continued lack of understanding about how cacheability and cookies coexist.

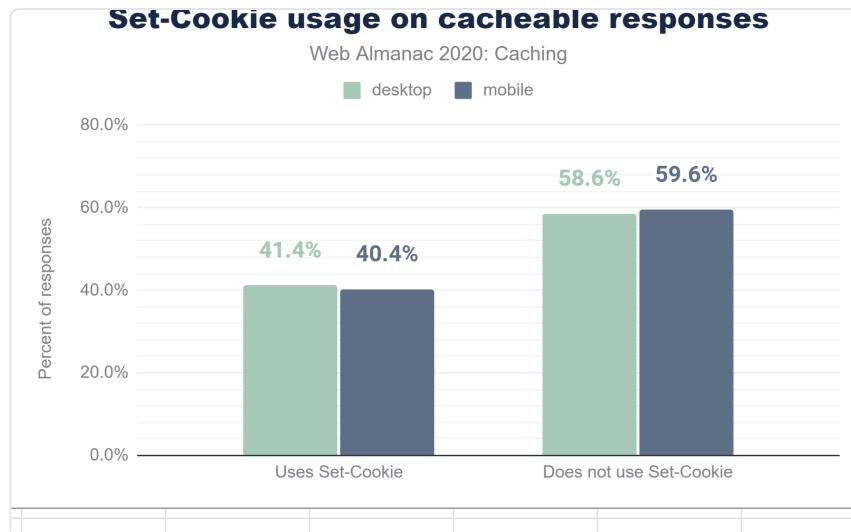


Figure 20.13. `Set-Cookie` in cacheable responses.

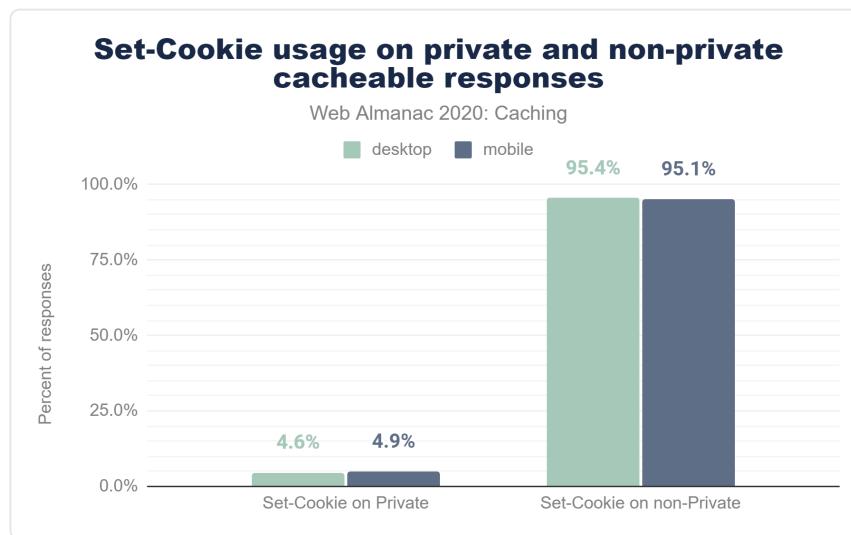


Figure 20.14. `Set-Cookie` in `private` and non private cacheable responses.

Service workers

Service workers are a feature of HTML5 that allow front-end developers to specify scripts that should run outside the 'normal' request/response flow of web pages, communicating with the web page via messages. Common uses of service workers are for background synchronization and push notifications and, obviously, for caching - and browser support has been rapidly growing for them.

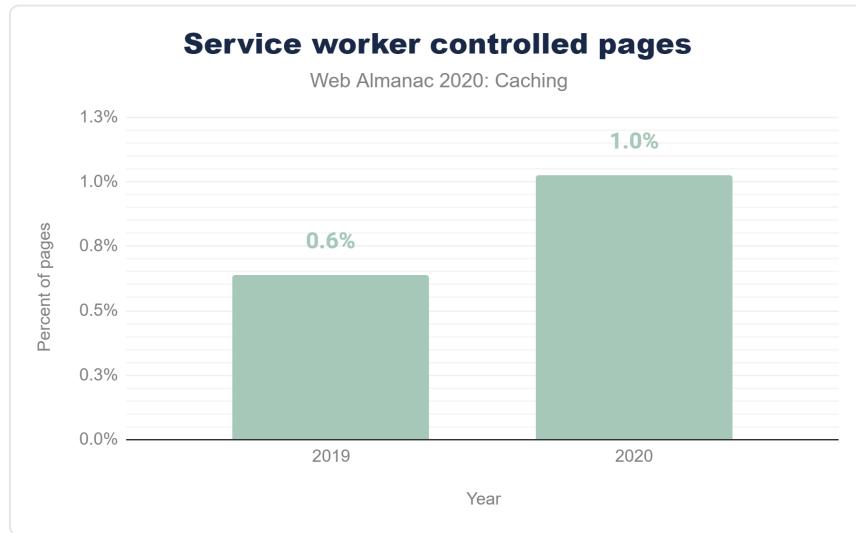


Figure 20.15. Growth in service worker controlled pages from 2019.

Adoption is just at 1% of websites, but it has been steadily increasing since July 2019. The Progressive Web App chapter discusses this more, including the fact that it is used a lot more than this graph suggests due to its usage on popular sites, which are only counted once in the above graph.

In the table below, you can see that out of a total of 6,225,774 websites, only 64,373 (1%) have implemented a service worker.

Sites not using service workers	Sites using service workers	Total sites
6,225,774	64,373	6,290,147

Figure 20.16. Number of websites using service workers.

If we break this out by HTTP vs HTTPS, then this gets even more interesting. Even though

HTTPS is a requirement for using service workers, the following table shows that 1,469 of the sites using them are served over HTTP.

HTTP Sites	HTTPS Sites	Total Sites
1,469	62,904	64,373

Figure 20.17. Number of websites using service workers by HTTP/HTTPS.

What type of content are we caching?

As we have seen, a cacheable resource is stored by the browser for a period of time and is available for reuse on subsequent requests. Across all HTTP(S) requests, 90.8% of responses are considered cacheable, meaning that a cache is permitted to store them. Out of these,

- 4.2% of requests have a TTL of 0 seconds, which causes the object to be added to cache, but immediately marked as stale, requiring revalidation.
- 28.2% are cached heuristically because of a lack of either a `Cache-Control` or `Expires` header.
- 59.4% are cached for more than 0 seconds.

The remaining 9.2% of responses are not permitted to be stored in browser caches - typically because of `Cache-Control: no-store`.

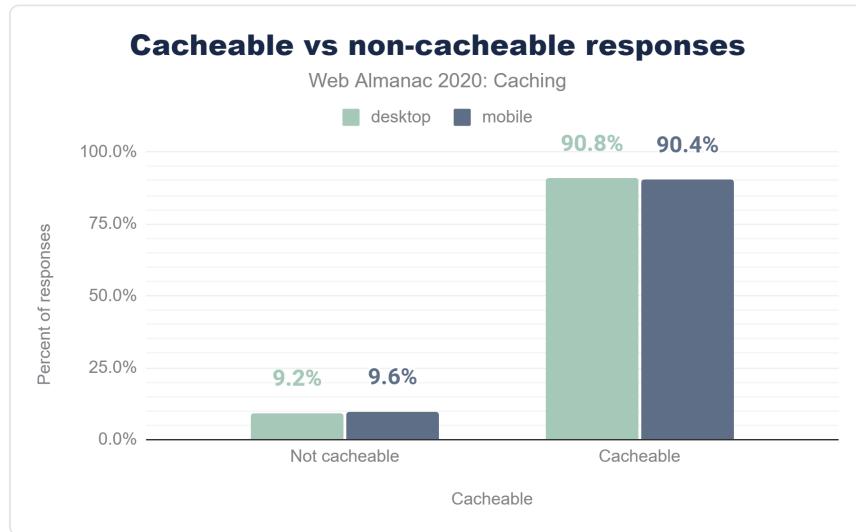


Figure 20.18. Distribution of cacheable and non-cacheable responses.

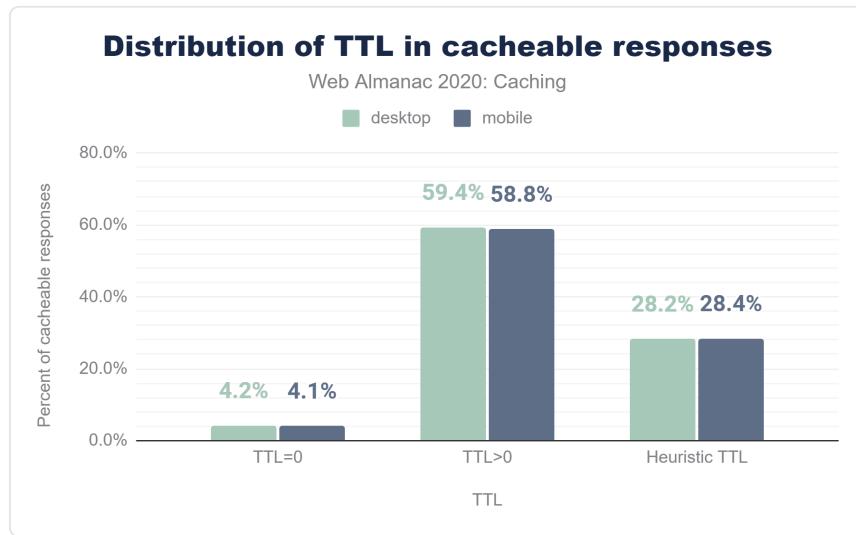


Figure 20.19. Distribution of TTL in cacheable responses.

The table below details the cache TTL values for desktop requests by type. Most content types are being cached, however CSS resources are consistently cached with high TTLs.

Type	10	25	50	75	90
audio	6	6	240	720	8,760
css	24	24	720	8,760	8,760
font	720	8,760	8,760	8,760	8,760
html	0	1	336	8,760	87,600
image	4	168	720	8,760	8,766
other	0	1	30	240	8,760
script	0	2	720	8,760	8,760
text	0	1	6	6	720
video	6	12	336	336	8,760
xml	0	24	24	24	8,760

Figure 20.20. Desktop cache TTL percentiles by resource type.

While most of the median TTLs are high, the lower percentiles highlight some of the missed caching opportunities. For example, the median TTL for images is 720 hours (1 month); however the 25th percentile is just 168 hours (1 week) and the 10th percentile has dropped to just a few hours. Compare this with fonts, which have a very high TTL of 8760 hours (1 year) all the way down to the 25th percentile, with even the 10th percentile showing a TTL of 1 month.

By exploring the cacheability by content type in more detail in figure below, we can see that while fonts, video and audio, and CSS files are browser cached at close to 100% (which makes sense, since these files are typically very static), approximately one third of all HTML responses are considered non-cacheable.

Additionally, 13.6% of images and scripts are non-cacheable. There is likely some room for improvement here, since no doubt some of these objects are also static and could be cached at a higher rate - remember: *cache as much as you can for as long as you can!*

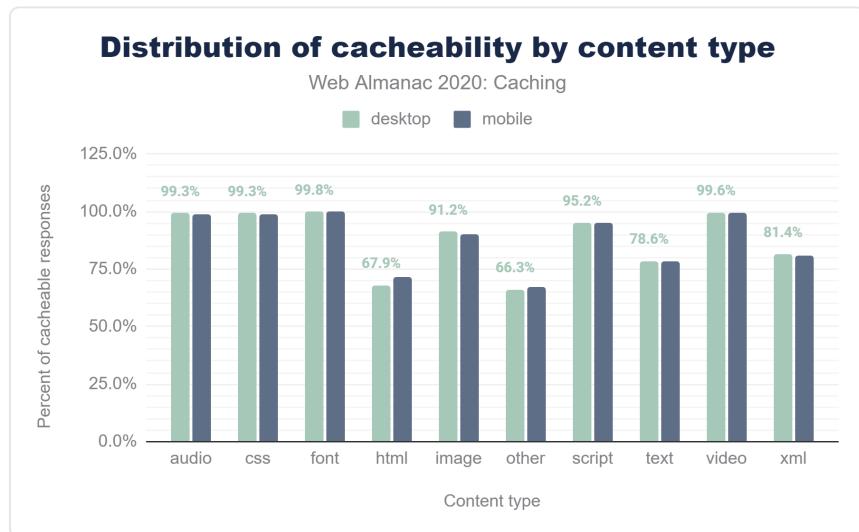


Figure 20.21. Distribution of cacheability by content type.

How do cache TTLs compare to resource age?

So far we've talked about how servers tell a client what is cacheable, and how long it has been cached for. When designing cache rules, it is also important to understand how old the content you are serving is.

When you (the server) are selecting a cache TTL to specify in response headers to send back to the client, ask yourself: "how often am I updating these assets?" and "what is their content sensitivity?". For example, if a hero image is going to be modified infrequently, then it could be cached with a very long TTL. By contrast, if a JavaScript file will change frequently, then either it should be versioned (for instance, by using an ETag or with a unique query string) and cached with a long TTL or it should be cached with a much shorter TTL.

The graphs below illustrate the relative age of resources by content type. Some of the interesting observations in this data are:

- First party HTML is the content type with the shortest age, with 42.5% of the requests having an age less than a week. In most of the other content types, third party content has a smaller resource age than first party content.
- Some of the longest aged first party content on the web, with age eight weeks or more, are the traditionally cacheable objects like images (78.3%), scripts (68.6%), CSS (74.1%), web fonts (79.3%), audio (77.9%) and video (78.6%).

- There is a significant gap in some first vs. third party resources having an age of more than a week. 93.5% of first party CSS are older than one week compared to 51.5% of 3rd party CSS, which are older than one week.

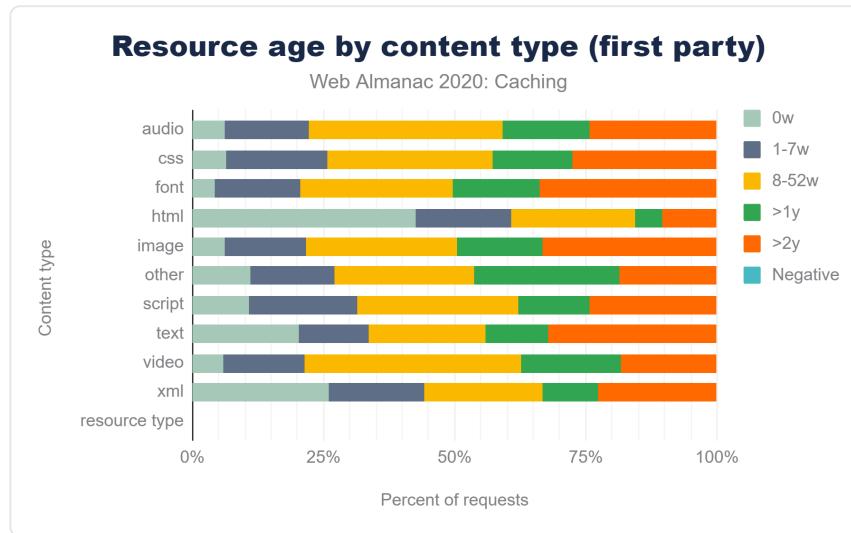


Figure 20.22. Resource age by Content Type (1st Party).

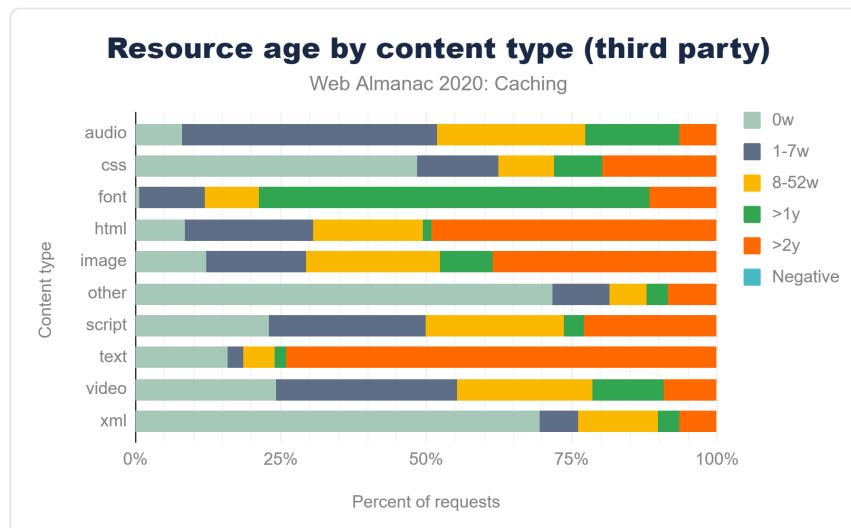


Figure 20.23. Resource age by Content Type (3rd Party).

By comparing a resource's cacheability to its age, we can determine if the TTL is appropriate or

too low.

For example, the resource served below on 18 Oct 2020 was last modified on 30 Aug 2020, which means that it was well over a month old at the time of delivery - this indicates that it is an object which does not change frequently. However, the `Cache-Control` header says that the browser can cache it for only 86,400 seconds (one day). This is a case where a longer TTL might be appropriate, to avoid the browser needing to re-request it (even conditionally) - especially if the website is one that a user might visit multiple times over the course of several days.

```
> HTTP/1.1 200
> Date: Sun, 18 Oct 2020 19:36:57 GMT
> Content-Type: text/html; charset=utf-8
> Content-Length: 3052
> Vary: Accept-Encoding
> Last-Modified: Sun, 30 Aug 2020 16:00:30 GMT
> Cache-Control: public, max-age=86400
```

Overall, 60.7% of resources served on the web have a cache TTL that could be considered too short compared to its content age. Furthermore, the median delta between the TTL and age is 25 days - again, an indication of significant under-caching.

When we break this out by first party vs third party in the following table, we can see that almost two-thirds (61.6%) of first-party resources can benefit from a longer TTL. This clearly highlights a need to spend extra attention focusing on what is cacheable, and then ensuring that caching is configured correctly.

Client	1st party	3rd party	Overall
desktop	61.6%	59.3%	60.7%
mobile	61.8%	57.9%	60.2%

Figure 20.24. Percent of requests with short TTLs.

Identifying caching opportunities

Google's Lighthouse tool enables users to run a series of audits against web pages, and the cache policy audit evaluates whether a site can benefit from additional caching. It does this by comparing the content age (via the `Last-Modified` header) to the cache TTL and estimating

the probability that the resource would be served from cache. Depending on the score, you may see a caching recommendation in the results, with a list of specific resources that could be cached.

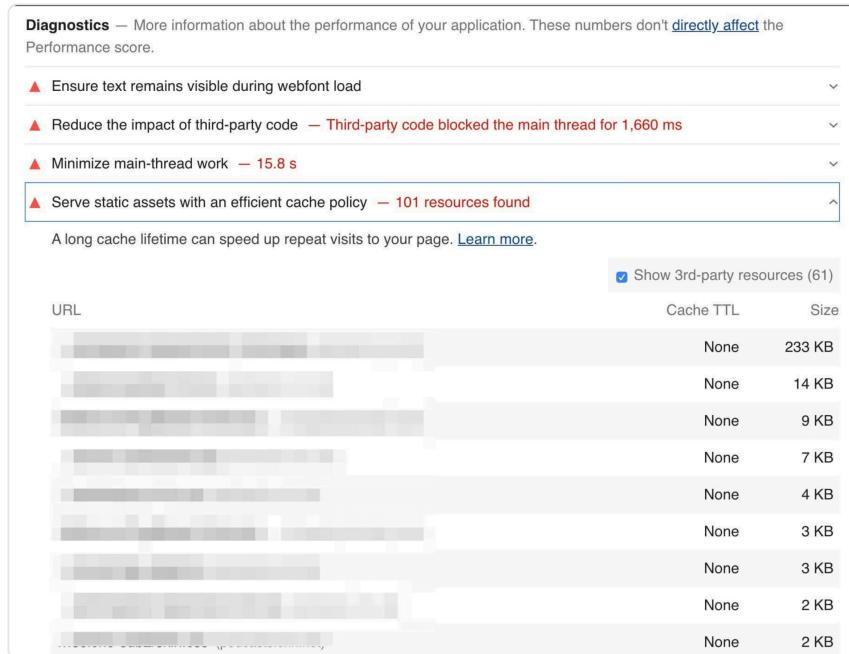


Figure 20.25. Lighthouse report highlighting potential cache policy improvements.

Lighthouse computes a score for each audit, ranging from 0% to 100%, and those scores are then factored into the overall scores. The caching score is based on potential byte savings. When we examine the Lighthouse results, we can get a perspective of how many sites are doing well with their cache policies.

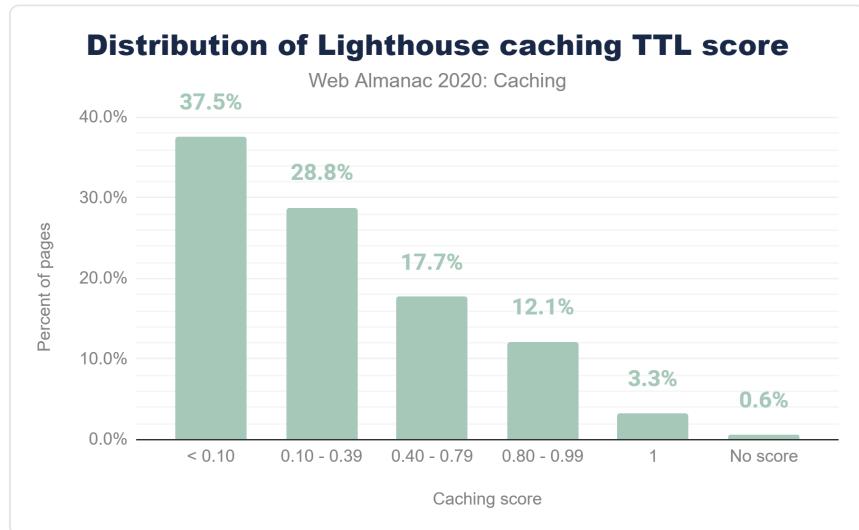


Figure 20.26. Distribution of Lighthouse audit scores for the `uses-long-cache-ttl` for mobile web pages.

Only 3.3% of sites scored a 100%, meaning that the vast majority of sites can benefit from some cache optimizations. Approximately two-thirds of sites score below 40%, with almost one-third of sites scoring less than 10%. Based on this, there is a significant amount of under-caching, resulting in excess requests and bytes being served across the network.

Lighthouse also indicates how many bytes could be saved on repeat views by enabling a longer cache policy. Of the sites that could benefit from additional caching, more than one-fifth can reduce their page weight by over 2MB!

Distribution of potential byte savings from caching

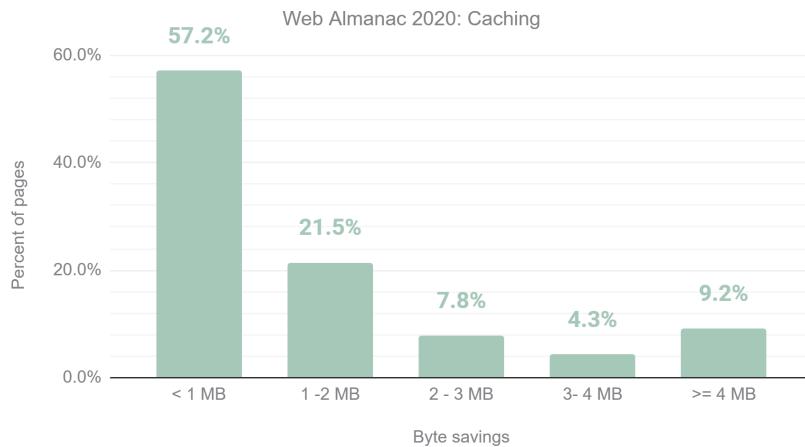


Figure 20.27. Distribution of potential byte savings from the Lighthouse caching audit.

Conclusion

Caching is an incredibly powerful feature that allows browsers, proxies and other intermediaries (such as CDNs) to store web content and serve it to end users. The performance benefits of this are significant, since it reduces round trip times and minimizes costly network requests.

Caching is also a very complex topic, and one that is often left until late in the development cycle (due to requirements by site developers to see the very latest version of a site while it is still being designed), then being added in at the last minute. Additionally, caching rules are often defined once and then never changed, even as the underlying content on a site changes. Frequently a default value is chosen without careful consideration.

To correctly cache objects, there are numerous HTTP response headers that can convey freshness as well as validate cached entries, and `Cache-Control` directives provide a tremendous amount of flexibility and control.

Many object types and content that are typically considered to be uncacheable can actually be cached (remember: *cache as much as you can!*) and many objects are cached for too short a period of time, requiring repeated requests and revalidation (remember: *cache for as long as you can!*). However, website developers should be cautious about the additional opportunities for mistakes that come with over-caching content.

If the site is intended to be served through a CDN, additional opportunities for caching at the CDN to reduce server load and provide faster response to end-users should be considered, along with the related risks of accidentally caching private information, such as cookies.

However, 'powerful' and 'complex' do not imply 'difficult' - like most everything else, caching is controlled by rules which can be defined fairly easily to provide the best mix of cacheability and privacy. Regularly auditing your site to ensure that cacheable resources are cached appropriately is recommended, and tools like Lighthouse do an excellent job of helping to simplify such an analysis.

Authors



Rory Hewitt

@roryhewitt3 roryhewitt roryhewitt <https://romche.com/>

Enterprise Architect at Akamai⁶¹, who is passionate about performance. A British ex-patriate, he has lived in San Francisco for more than twenty years. In his spare time, he's a long-distance adventure motorcyclist, snowboarder and boxer/karateka. He likes being known as a troublemaker. Most importantly, he's a father and husband and the owner of Luna the cat.



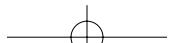
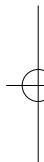
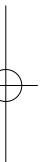
Raghu Ramakrishnan

raghuramakrishnan71

Enterprise architect at Tata Consultancy Services⁶², working on large digital transformation programs in the public sector. A technology enthusiast with a special interest in performance engineering. An avid traveler, intrigued by astronomy, history, biology, and advancements in medicine. A strong follower of the 47th verse, Chapter 2 of Bhagavad Gita "karmany-evādhikāras te mā phaleśhu kadāchana" meaning "You have a right to perform your prescribed duty, but you are not entitled to the fruits of action."

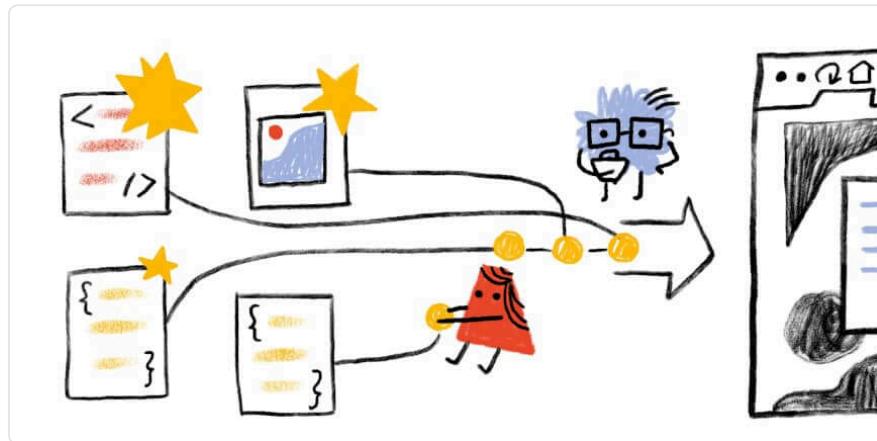
61. <https://www.akamai.com/>

62. <https://www.tcs.com/>



Part IV Chapter 21

Resource Hints



Written by Leonardo Zizzamia

Reviewed by Jessica Nicolet, Patrick Meenan, Giovanni Punti, Minko Gechev, and notwillk

Analyzed by Katie Hempenius

Introduction

Over the past decade resource hints have become essential primitives that allow developers to improve page performance and therefore the user experience.

Preloading resources and having browsers apply some intelligent prioritization is something that was actually started way back in 2009 by IE8 with something called the preloader. In addition to the HTML parser, IE8 had a lightweight look-ahead preloader that scanned for tags that could initiate network requests (`<script>`, `<link>`, and ``).

Over the following years, browser vendors did more and more of the heavy lifting, each adding their own special sauce for how to prioritize resources. But it's important to understand that the browser alone has some limitations. As developers however, we can overcome these limits by making good use of resource hints and help decide how to prioritize resources, determining which should be fetched or preprocessed to further boost page performance.

In particular we can mention a few of the victories resource hints achieved/made in the last

year:

- CSS-Tricks web fonts showing up faster on a 3G first render.
- Wix.com using resource hints got 10% improvement for FCP.
- Ironmongerydirect.co.uk used preconnect to improve product image loading by 400ms at the median and greater than 1s at the 95th percentile.
- Facebook.com used preload for faster navigation.

Let's take a look at most predominant resource hints supported by most browsers today: `dns-prefetch`, `preconnect`, `preload`, `prefetch`, and native lazy loading.

When working with each individual hint we advise to always measure the impact before and after in the field, by using libraries like WebVitals, Perfume.js, or any other utility that supports the Web Vitals metrics.

`dns-prefetch`

`dns-prefetch` helps resolve the IP address for a given domain ahead of time. As the oldest resource hint available, it uses minimal CPU and network resources compared to `preconnect`, and helps the browser to avoid experiencing the "worst-case" delay for DNS resolution, which can be over 1 second.

```
<link rel="dns-prefetch" href="https://www.googletagmanager.com/">
```

Be mindful when using `dns-prefetch` as even if they are lightweight to do it's easy to exhaust browser limits for the number of concurrent in-flight DNS requests allowed (Chrome still has a limit of 6).

`preconnect`

`preconnect` helps resolve the IP address and open a TCP/TLS connection for a given domain ahead of time. Similar to `dns-prefetch` it is used for any cross-origin domain and helps the browser to warm up any resources used during the initial page load.

```
<link rel="preconnect" href="https://www.googletagmanager.com/">
```

Be mindful when you use `preconnect`:

- Only warm up the most frequent and significant resources.
- Avoid warming up origins used too late in the initial load.
- Use it for no more than three origins because it can have CPU and battery cost.

Lastly, `preconnect` is not available for Internet Explorer or Firefox, and using `dns-prefetch` as a fallback is highly advised.

preload

The `preload` hint initiates an early request. This is useful for loading important resources that would otherwise be discovered late by the parser.

```
<link rel="preload" href="style.css" as="style">
<link rel="preload" href="main.js" as="script">
```

Be mindful of what you are going to `preload`, because it can delay the download of other resources, so use it only for what is most critical to help you improve the Largest Contentful Paint (LCP). Also, when used on Chrome, it tends to over-prioritize `preload` resources and potentially dispatches preloads before other critical resources.

Lastly, if used in a HTTP response header, some CDN's will also automatically turn a `preload` into a HTTP/2 push which can over-push cached resources.

prefetch

The `prefetch` hint allows us to initiate low-priority requests we expect to be used on the next navigation. The hint will download the resources and drop it into the HTTP cache for later usage. Important to notice, `prefetch` will not execute or otherwise process the resource, and to execute it the page will still need to call the resource by the `<script>` tag.

```
<link rel="prefetch" as="script" href="next-page.bundle.js">
```

There are a variety of ways to implement a resource's prediction logic, it could be based on signals like user mouse movement, common user flows/journeys, or even based on a combination of both on top of machine learning.

Be mindful, depending on the quality of HTTP/2 prioritization of the CDN used, `prefetch`

prioritization could either improve performance or make it slower, by over prioritizing `prefetch` requests and taking away important bandwidth for the initial load. Make sure to double check the CDN you are using and adapt to take into consideration some of the best practices shared in Andy Davies's article.

Native lazy loading

The native lazy loading hint is a native browser API for deferring the load of offscreen images and iframes. By using it, assets that are not needed during the initial page load will not initiate a network request, this will reduce data consumption and improve page performance.

```

```

Be mindful Chromium's implementation of lazy-loading thresholds logic historically has been quite conservative, keeping the offscreen limit to 3000px. During the last year the limit has been actively tested and improved on to better align developer expectations, and ultimately moving the thresholds to 1250px. Also, there is no standard across the browsers and no ability for web developers to override the default thresholds provided by the browsers, yet.

Resource hints

Based on the HTTP Archive, let's jump into analyzing the 2020 trends, and compare the data with the previous 2019 dataset.

Hints adoption

More and more web pages are using the main resource hints, and in 2020 we are seeing the adoption remains consistent between desktop & mobile.

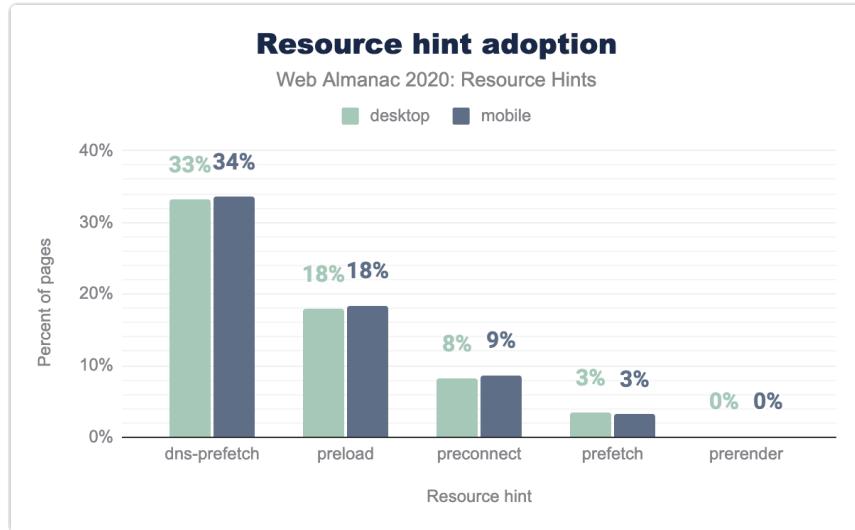


Figure 21.1. Adoption of resource hints.

The relative popularity of `dns-prefetch` with 33% adoption compared with other resource hints is unsurprising as it first appeared in 2009, and has the widest support out of all major resource hints.

Compared to 2019 the `dns-prefetch` had a 4% increase in Desktop adoption. We saw a similar increase for `preconnect` as well. One key reason this was the largest growth between all hints, is the clear and useful advice the Lighthouse audit is giving on this matter. Starting from this year's report we also introduce how the latest dataset performs against Lighthouse recommendations.

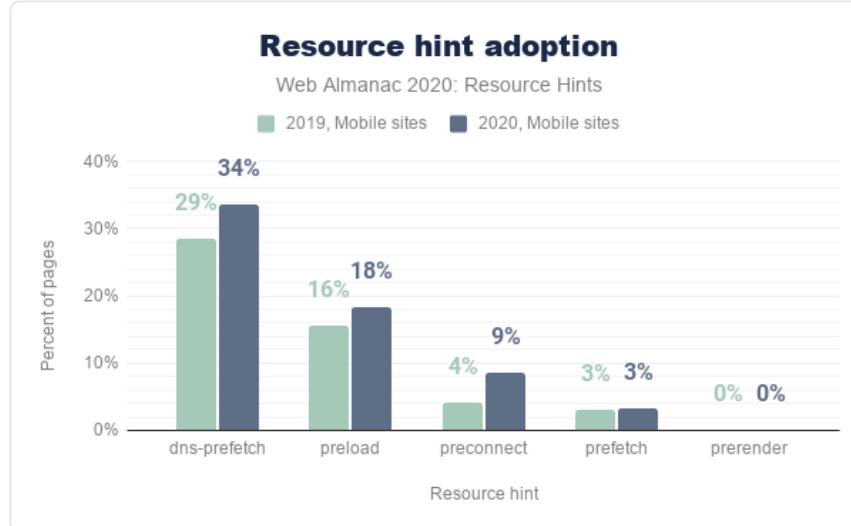


Figure 21.2. Adoption of resource hints 2019 vs 2020.

`preload` usage has had a slower growth with only a 2% increase from 2019. This could be in part because it requires a bit more specification. While you only need the domain to use `dns-prefetch` and `preconnect`, you must specify the resource to use `preload`. While `dns-prefetch` and `preconnect` are reasonably low risk, though still can be abused, `preload` has a much greater potential to actually damage performance if used incorrectly.

`prefetch` is used by 3% of sites on Desktop, making it the least widely used resource hint. This low usage may be explained by the fact that `prefetch` is useful for improving subsequent, rather than current, page loads. Thus, it will be overlooked if a site is only focused on improving its landing page, or the performance of the first page viewed. In the coming years with a more clear definition on what to measure for improving subsequent page experience, it could help teams prioritize `prefetch` adoption with clear performance quality goals to reach.

Hints per page

Across the board developers are learning how to better use resource hints, and compared to 2019 we've seen an improved use of `preload`, `prefetch`, and `preconnect`. For expensive operations like preload and preconnect the median usage on desktop decreased from 2 to 1. We have seen the opposite for loading future resources with a lower priority with `prefetch`, with an increase from 1 to 2 in median per page.

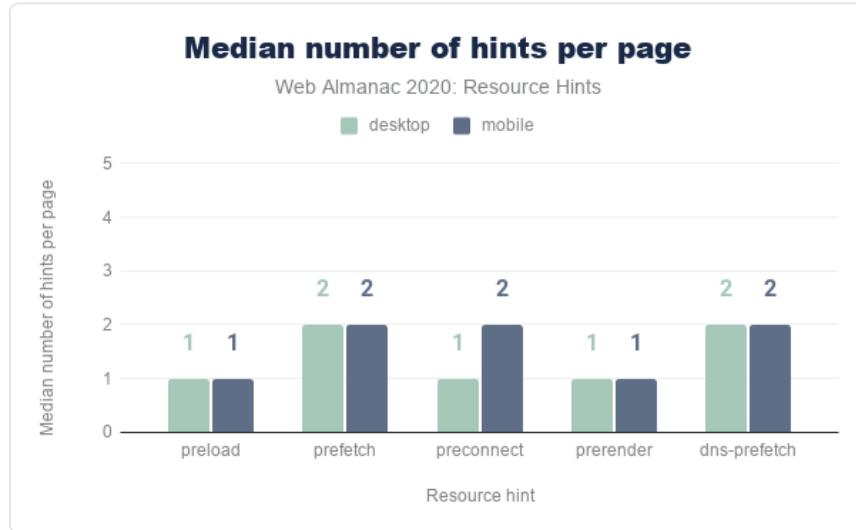


Figure 21.3. Median number of hints per page.

Resource hints are most effective when they're used selectively ("when everything is important, nothing is"). Having a more clear definition of what resources help improve critical rendering, versus future navigation optimizations, can move the focus away from using `preconnect` and more towards `prefetch` by shifting some of the resource prioritization and freeing up bandwidth for what most helps the user at first.

However, this hasn't stopped some misuse of the `preload` hint, since in one instance we discovered a page dynamically adding the hint and causing an infinite loop that created over 20k new preloads.

20,931

Figure 21.4. The most preload hints on a single page.

As we create more and more automation with resource hints, be cautious when dynamically injecting preload hints - or any elements for that matter!

The `as` attribute

With `preload` and `prefetch`, it's crucial to use the `as` attribute to help the browser prioritize the resource more accurately. Doing so allows for proper storage in the cache for future requests, applying the correct Content Security Policy (CSP), and setting the correct `Accept` request headers.

With `preload` many different content-types can be preloaded and the full list follows the recommendations made in the Fetch spec. The most popular is the `script` type with 64% usage. This is likely related to a large group of sites built as Single Page Apps that need the main bundle as soon as possible to start downloading the rest of their JS dependencies. Subsequent usage comes from font at 8%, style at 5%, image at 1%, and fetch at 1%.

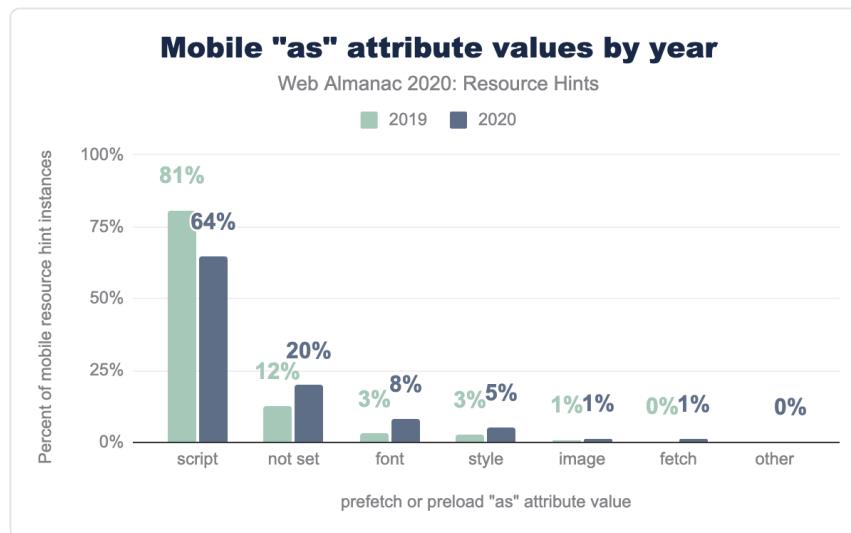


Figure 21.5. Mobile `as` attribute values by year.

Compared to the trend in 2019, we've seen rapid growth in font and style usage with the `as` attribute. This is likely related to developers increasing the priority of critical CSS and also combining `preload` fonts with `display:optional` to improve Cumulative Layout Shift (CLS).

Be mindful that omitting the `as` attribute, or having an invalid value will make it harder for the browser to determine the correct priority and in some cases, such as scripts, can even cause the resource to be fetched twice.

The `crossorigin` attribute

With `preload` and `preconnect` resources that have CORS enabled, such as fonts, it's important to include the `crossorigin` attribute, in order for the resource to be properly used. If the `crossorigin` attribute is absent, the request will follow the single-origin policy thereby making the use of preload useless.

16.96%

Figure 21.6. The percent of elements with `preload` that use `crossorigin`.

The latest trends show that 16.96% of elements that `preload` also set `crossorigin` and load in anonymous (or equivalent) modes, and only 0.02% utilize the `use-credentials` case. This rate has increased in conjunction with the increase in font-preloading, as mentioned earlier.

```
<link rel="preload" href="ComicSans.woff2" as="font" type="font/woff2" crossorigin>
```

Be mindful that fonts preloaded without the `crossorigin` attribute will be fetched twice!

The `media` attribute

When it's time to choose a resource for use with different screen sizes, reach for the `media` attribute with `preload` to optimize your media queries.

```
<link rel="preload" href="a.css" as="style" media="only screen and (min-width: 768px)">
<link rel="preload" href="b.css" as="style" media="screen and (max-width: 430px)">
```

Seeing over 2,100 different combinations of media queries in the 2020 dataset encourages us to consider how wide the variance is between concept and implementation of responsive design from site to site. The ever popular `767px/768px` breakpoints (as popularized by

Bootstrap amongst others) can be seen in the data.

Best practices

Using resource hints can be confusing at times, so let's go over some quick best practices to follow based on Lighthouse's automated audit.

To safely implement `dns-prefetch` and `preconnect` make sure to have them in separate `links` tags.

```
<link rel="preconnect" href="http://example.com">
<link rel="dns-prefetch" href="http://example.com">
```

Implementing a `dns-prefetch` fallback in the same `<link>` tag causes a bug in Safari that cancels the `preconnect` request. Close to 2% of pages (~40k) reported the issue of both `preconnect` & `dns-prefetch` in a single resource.

In the case of "Preconnect to required origins" audit, we saw only 19.67% of pages passing the test, creating a large opportunity for thousands of websites to start using `preconnect` or `dns-prefetch` to establish early connections to important third-party origins.

A large, bold, blue percentage value of 19.67% is displayed, representing the percentage of pages that pass the preconnect Lighthouse audit.

Figure 21.7. The percent of pages that pass the `preconnect` Lighthouse audit.

Lastly, running Lighthouse's "Preload key requests" audit resulted in 84.6% of pages passing the test. If you are looking to use `preload` for the first time, remember, fonts and critical scripts are a good place to start.

Native Lazy Loading

Now let's celebrate the first year of the Native Lazy Loading API, which at the time of publishing already has over 72% browser support. This new API can be used to defer the load of below-the-fold iframes and images on the page until the user scrolls near them. This can reduce data usage, memory usage, and helps speed up above-the-fold content. Opting-in to lazy load is as simple as adding `loading=lazy` on `<iframe>` or `` elements.

4.02%

Figure 21.8. The percent of pages using native lazy loading.

Adoption is still in its early days, especially with the official thresholds earlier this year being too conservative, and only recently aligning with developer expectations. With almost 72% of browsers supporting native image/source lazy loading, this is another area of opportunity especially for pages looking to improve data usage and performance on low-end devices.

Running Lighthouse's "Defer offscreen images" audit resulted in 68.65% of pages passing the test. For those pages there is an opportunity to lazy-load images after all critical resources have finished loading.

Be mindful to run the audit on both desktop and mobile as images may move off screen when the viewport changes.

Predictive prefetching

Combining `prefetch` with machine learning can help improve the performance of subsequent page(s). One solution is Guess.js which made the initial breakthrough in predictive-prefetching, with over a dozen websites already using it in production.

Predictive prefetching is a technique that uses methods from data analytics and machine learning to provide a data-driven approach to prefetching. Guess.js is a library that has predictive prefetching support for popular frameworks (Angular, Nuxt.js, Gatsby, and Next.js) and you can take advantage of it today. It ranks the possible navigations from a page and prefetches only the JavaScript that is likely to be needed next.

Depending on the training set, the prefetching of Guess.js comes with over 90% accuracy.

Overall, predictive prefetching is still uncharted territory, but combined with prefetching on mouse over and Service Worker prefetching, it has great potential to provide instant experiences for all users of the website, while saving their data.

HTTP/2 Push

HTTP/2 has a feature called "server push" that can potentially improve page performance when

your product experiences long Round Trip Times(RTTs) or server processing. In brief, rather than waiting for the client to send a request, the server preemptively pushes a resource that it predicts the client will request soon afterwards.

75.36%

Figure 21.9. The percent of HTTP/2 Push pages using `preload`/`nopush`.

HTTP/2 Push is often initiated through the `preload` link header. In the 2020 dataset we have seen 1% of mobile pages using HTTP/2 Push, and of those 75% of preload header links use the `nopush` option in the page request. This means that even though a website is using the `preload` resource hint, the majority prefer to use just this and disable HTTP/2 pushing of that resource.

It's important to mention that HTTP/2 Push can also damage performance if not used correctly which probably explains why it is often disabled.

One solution to this, is to use the PRPL Pattern which stands for **P**ush (or preload) the critical resources, **R**ender the initial route as soon as possible, **P**re-cache remaining assets, and **L**azy-load other routes and non-critical assets. This is possible only if your website is a Progressive Web App and uses a Service Worker to improve the caching strategy. By doing this, all subsequent requests never even go out to the network and so there's no need to push all the time and we still get the best of both worlds.

Service Workers

For both `preload` and `prefetch` we've had an increase in adoption when the page is controlled by a Service Worker. This is because of the potential to both improve the resource prioritization by preloading when the Service Worker is not active yet and intelligently prefetching future resources while letting the Service Worker cache them before they're needed by the user.

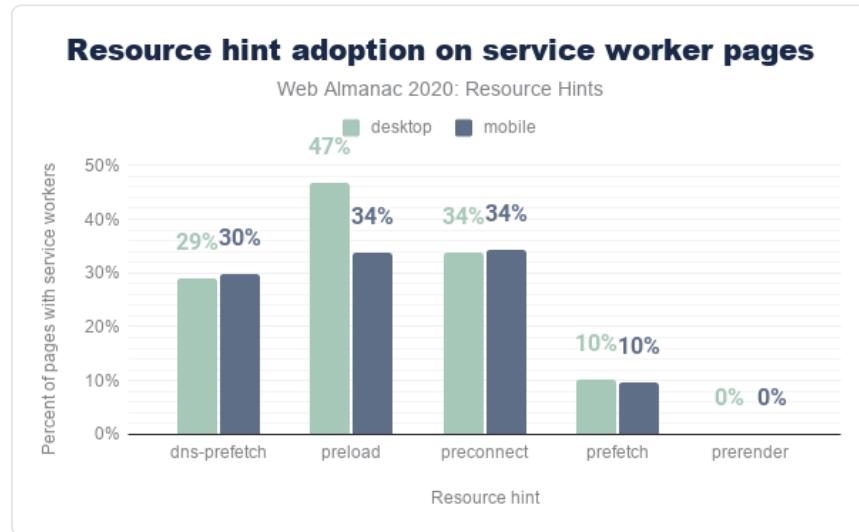


Figure 21.10. Resource hint adoption on Service Worker pages.

For `preload` on desktop we have an outstanding 47% rate of adoption and `prefetch` a 10% rate of adoption. In both cases the data is much higher compared to average adoption without a Service Worker.

As mentioned earlier, the PRPL Pattern will play a significant role in the coming years in how we combine resource hints with the Service Worker caching strategy.

Future

Let's dive into a couple of experimental hints. Very close to release we have Priority Hints, which is actively experimented with in the web community. We also have the 103 Early Hints in HTTP/2, which is still in early inception and there are a few players like Chrome and Fastly collaborating for upcoming test trials.

Priority hints

Priority hints are an API for expressing the fetch priority of a resource: high, low, or auto. They can be used to help deprioritize images (e.g. inside a Carousel), re-prioritize scripts, and even help de-prioritize fetches.

This new hint can be used either as an HTML tag or by changing the priority of fetch requests

via the `importance` option, which takes the same values as the HTML attribute.

```
<!-- We want to initiate an early fetch for a resource, but also
deprioritize it -->
<link rel="preload" href="/js/script.js" as="script"
importance="low">

<!-- An image the browser assigns "High" priority, but we don't
actually want that. -->

```

With `preload` and `prefetch`, the priority is set by the browser depending on the type of resource. By using Priority Hints we can force the browser to change the default option.

0.77%

Figure 21.11. The percent of Priority Hint adoption on mobile.

So far only 0.77% websites adopted this new hint as Chrome is still actively experimenting, and at the time of this article's release the feature is on-hold.

The largest use is with script elements, which is unsurprising as the number of JS primary and third-party files continues to grow.

16%

Figure 21.12. The percent of mobile resources with a hint that use the "low" priority.

The data shows us that 83% of resources using Priority Hints use a "high" priority on mobile, but something we should pay even more attention to is the 16% of resources with "low" priority.

Priority hints have a clear advantage as a tool to prevent wasteful loading via the "low" priority by helping the browser decide what to de-prioritize and giving back significant CPU and bandwidth to complete critical requests first, rather than as a tactic to try to get resources loaded more quickly with the "high" priority.

103 Early Hints in HTTP/2

Previously we mentioned that HTTP/2 Push could actually cause regression in cases where assets being pushed were already in the browser cache. The 103 Early Hints proposal aims to provide similar benefits promised by HTTP/2 push. With an architecture that is potentially 10x simpler, it addresses the long RTT's or server processing without suffering from the known worst-case issue of unnecessary round trips with server push.

As of right now you can follow the conversation on Chromium with issues 671310, 1093693, and 1096414.

Conclusion

During the past year resource hints increased in adoption, and they have become essential APIs for developers to have more granular control over many aspects of resource prioritizations and ultimately, user experience. But let's not forget that these are hints, not instructions and unfortunately the Browser and the network will always have the final say.

Sure, you can slap them on a bunch of elements, and the browser may do what you're asking it to. Or it may ignore some hints and decide the default priority is the best choice for the given situation. In any case, make sure to have a playbook for how to best use these hints:

- Identify key pages for the user experience.
- Analyze the most important resources to optimize.
- Adopt the PRPL Pattern when possible.
- Measure the performance experience before and after each implementation.

As a final note, let's remember that the web is for everyone. We must continue to protect it and stay focused on building experiences that are easy and frictionless.

We are thrilled to see that year after year we get incrementally closer to offering all the APIs required to simplify building a great web experience for everyone, and we can't wait to see what comes next.

Author



Leonardo Zizzamia

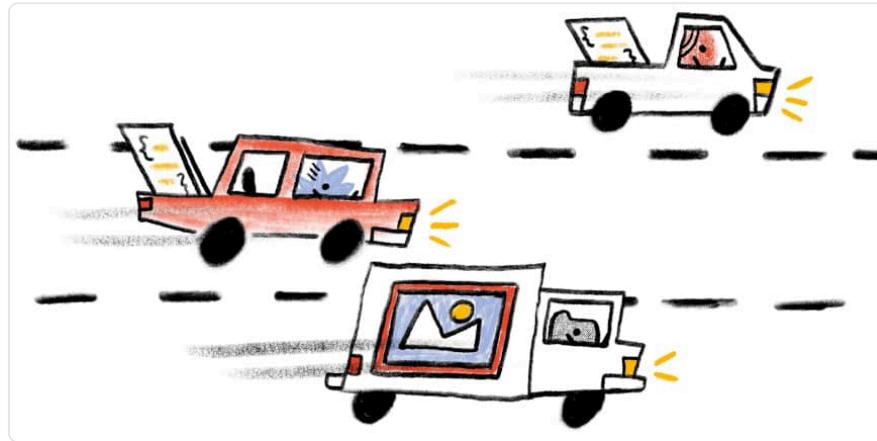
Twitter: [@Zizzamia](https://twitter.com/Zizzamia) GitHub: [Zizzamia](https://github.com/Zizzamia) URL: <https://twitter.com/zizzamia>

Leonardo is a Staff Software Engineer at Coinbase⁶³, leading web performance and growth initiatives. He curates the NGRome Conference⁶⁴. Leo also maintains the Perfume.js⁶⁵ library, which helps companies prioritize roadmaps and make better business decisions through performance analytics.

63. <https://www.coinbase.com/>
64. <https://ngrome.io>
65. <https://github.com/Zizzamia/perfume.js>

Part IV Chapter 22

HTTP/2



Written by Andrew Galloni, Robin Marx, and Mike Bishop

Reviewed by Lucas Pardue, Barry Pollard, and Sawood Alam

Analyzed by Greg Wolf

Introduction

HTTP is an application layer protocol designed to transfer information between networked devices and runs on top of other layers of the network protocol stack. After HTTP/1.x was released, it took over 20 years until the first major update, HTTP/2, was made a standard in 2015.

It didn't stop there: over the last four years, HTTP/3 and QUIC (a new latency-reducing, reliable, and secure transport protocol) have been under standards development in the IETF QUIC working group. There are actually two protocols that share the same name: "Google QUIC" ("gQUIC" for short), the original protocol that was designed and used by Google, and the newer IETF standardized version (IETF QUIC/QUIC). IETF QUIC was based on gQUIC, but has grown to be quite different in design and implementation. On October 21, 2020, draft 32 of IETF QUIC reached a significant milestone when it moved to Last Call. This is the part of the standardization process when the working group believes they are almost finished and requests a final review from the wider IETF community.

This chapter reviews the current state of HTTP/2 and gQUIC deployment. It explores how well some of the newer features of the protocol, such as prioritization and server push, have been adopted. We then look at the motivations for HTTP/3, describe the major differences between the protocol versions, and discuss the potential challenges in upgrading to a UDP-based transport protocol with QUIC.

HTTP/1.0 to HTTP/2

As the HTTP protocol has evolved, the semantics of HTTP have stayed the same; there have been no changes to the HTTP methods (such as GET or POST), status codes (200, or the dreaded 404), URIs, or header fields. Where the HTTP protocol has changed, the differences have been the wire-encoding and the use of features of the underlying transport.

HTTP/1.0, published in 1996, defined the text-based application protocol, allowing clients and servers to exchange messages in order to request resources. A new TCP connection was required for each request/response, which introduced overhead. TCP connections use a congestion control algorithm to maximize how much data can be in-flight. This process takes time for each new connection. This "slow-start" means that not all the available bandwidth is used immediately.

In 1997, HTTP/1.1 was introduced to allow TCP connection reuse by adding "keep-alives", aimed at reducing the total cost of connection start-ups. Over time, increasing website performance expectations led to the need for concurrency of requests. HTTP/1.1 could only request another resource after the previous response had completed. Therefore, additional TCP connections had to be established, reducing the impact of the keep-alive connections and further increasing overhead.

HTTP/2, published in 2015, is a binary-based protocol that introduced the concept of bidirectional streams between client and server. Using these streams, a browser can make optimal use of a single TCP connection to *multiplex* multiple HTTP requests/responses concurrently. HTTP/2 also introduced a prioritization scheme to steer this multiplexing; clients can signal a request priority that allows more important resources to be sent ahead of others.

HTTP/2 Adoption

The data used in this chapter is sourced from the HTTP Archive and tests over seven million websites with a Chrome browser. As with other chapters, the analysis is split by mobile and desktop websites. When the results between desktop and mobile are similar, statistics are presented from the mobile dataset. You can find more details on the Methodology page. When reviewing this data, please bear in mind that each website will receive equal weight regardless

of the number of requests. We suggest you think of this more as investigating the trends across a broad range of active websites.



Figure 22.1. HTTP/2 usage by request. (Source: HTTP Archive)

Last year's analysis of HTTP Archive data showed that HTTP/2 was used for over 50% of requests and, as can be seen, linear growth has continued in 2020; now in excess of 60% of requests are served over HTTP/2.

64%

Figure 22.2. The percentage of requests that use HTTP/2.

When comparing Figure 22.3 with last year's results, there has been a **10% increase in HTTP/2 requests** and a corresponding 10% decrease in HTTP/1.x requests. This is the first year that gQUIC can be seen in the dataset.

Protocol	Desktop	Mobile
HTTP/1.1	**34.47%	34.11%
HTTP/2	63.70%	63.80%
gQUIC	1.72%	1.71%

Figure 22.3. HTTP version usage by request.

*** As with last year's crawl, around 4% of desktop requests did not report a protocol version. Analysis shows these to mostly be HTTP/1.1 and we worked to fix this gap in our statistics for future crawls and analysis. Although we base the data on the August 2020 crawl, we confirmed the fix in the October 2020 data set before publication which did indeed show these were HTTP/1.1 requests and so have added them to that statistic in above table.*

When reviewing the total number of website requests, there will be a bias towards common third-party domains. To get a better understanding of the HTTP/2 adoption by server install, we will look instead at the protocol used to serve the HTML from the home page of a site.

Last year around 37% of home pages were served over HTTP/2 and 63% over HTTP/1. This year, combining mobile and desktop, it is a roughly equal split, with slightly more desktop sites being served over HTTP/2 for the first time, as shown in Figure 22.4.

Protocol	Desktop	Mobile
HTTP/1.0	0.06%	0.05%
HTTP/1.1	49.22%	50.05%
HTTP/2	49.97%	49.28%

Figure 22.4. HTTP version usage for home pages.

gQUIC is not seen in the home page data for two reasons. To measure a website over gQUIC, the HTTP Archive crawl would have to perform protocol negotiation via the alternative services header and then use this endpoint to load the site over gQUIC. This was not supported this year, but expect it to be available in next year's Web Almanac. Also, gQUIC is predominantly used for third-party Google tools rather than serving home pages.

The drive to increase security and privacy on the web has seen requests over TLS increase by over 150% in the last 4 years. Today, over 86% of all requests on mobile and desktop are encrypted. Looking only at home pages, the numbers are still an impressive 78.1% of desktop and 74.7% of mobile. This is important because HTTP/2 is only supported by browsers over

TLS. The proportion served over HTTP/2, as shown in Figure 22.5, has also increased by 10 percentage points from last year, from 55% to 65%.

Protocol	Desktop	Mobile
HTTP/1.1	36.05%	34.04%
HTTP/2	63.95%	65.96%

Figure 22.5. HTTP version usage for HTTPS home pages.

With over 60% of websites being served over HTTP/2 or gQUIC, let's look a little deeper into the pattern of protocol distribution for all requests made across individual sites.

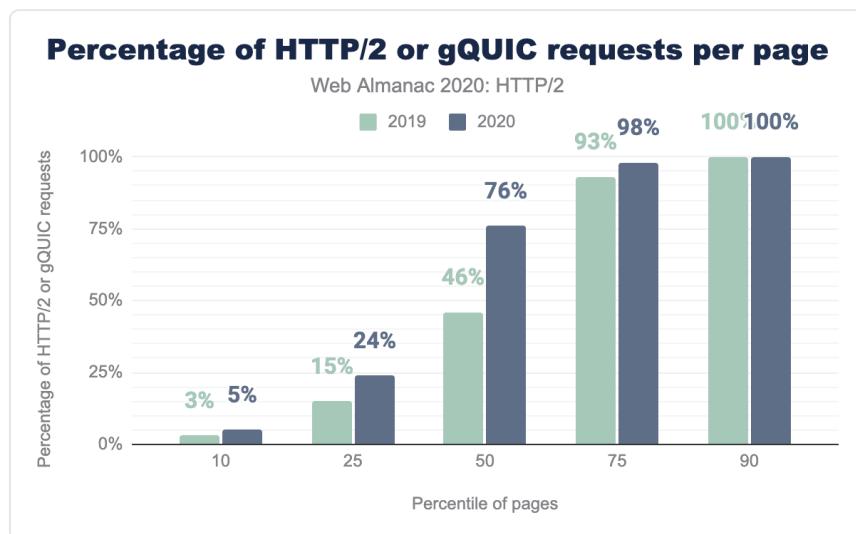


Figure 22.6. Compare the distribution of fraction of HTTP/2 requests per page in 2020 with 2019.

Figure 22.6 compares how much HTTP/2 or gQUIC is used on a website between this year and last year. The most noticeable change is that over half of sites now have 75% or more of their requests served over HTTP/2 or gQUIC compared to 46% last year. Less than 7% of sites make no HTTP/2 or gQUIC requests, while (only) 10% of sites are entirely HTTP/2 or gQUIC requests.

What about the breakdown of the page itself? We typically talk about the difference between first-party and third-party content. Third-party is defined as content not within the direct control of the site owner, providing functionality such as advertising, marketing or analytics. The definition of known third parties is taken from the third party web repository.

Figure 22.7 orders every website by the fraction of HTTP/2 requests for known third parties or first party requests compared to other requests. There is a noticeable difference as over 40% of all sites have no first-party HTTP/2 or gQUIC requests at all. By contrast, even the lowest 5% of pages have 30% of third-party content served over HTTP/2. This indicates that a large part of HTTP/2's broad adoption is driven by the third parties.

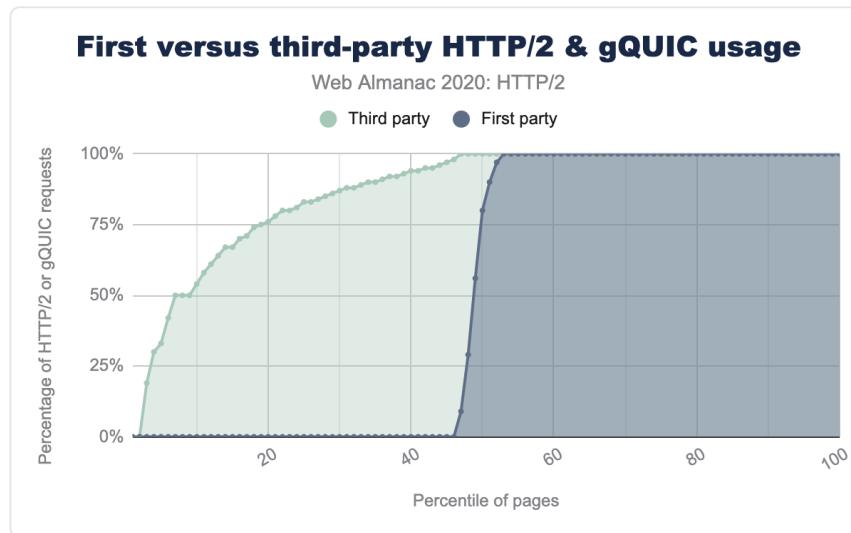


Figure 22.7. The distribution of the fraction of third-party and first-party HTTP/2 requests per page.

Is there any difference in which content-types are served over HTTP/2 or gQUIC? Figure 22.8 shows, for example, that 90% of websites serve 100% of third party fonts and audio over HTTP/2 or gQUIC, only 5% over HTTP/1.1 and 5% are a mix. The majority of third-party assets are either scripts or images, and are solely served over HTTP/2 or gQUIC on 60% and 70% of websites respectively.

Third-party HTTP/2 or gQUIC usage by content-type

Web Almanac 2020: HTTP/2

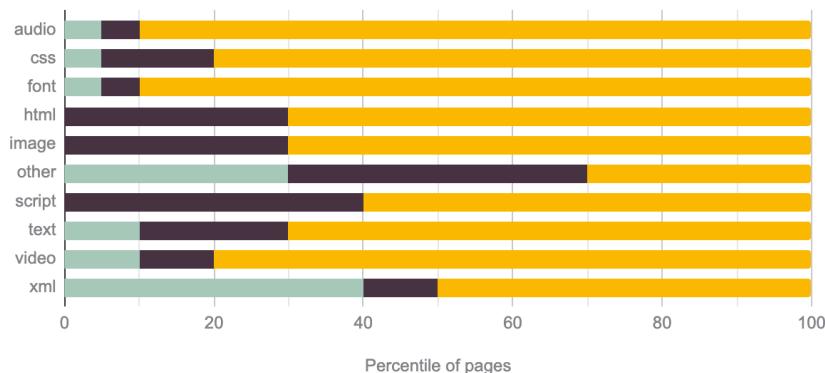



Figure 22.8. The fraction of known third-party HTTP/2 or gQUIC requests by content-type per website.

Ads, analytics, content delivery network (CDN) resources, and tag-managers are predominantly served over HTTP/2 or gQUIC as shown in Figure 22.9. Customer-success and marketing content is more likely to be served over HTTP/1.

Third-party HTTP/2 or gQUIC usage by category

Web Almanac 2020: HTTP/2

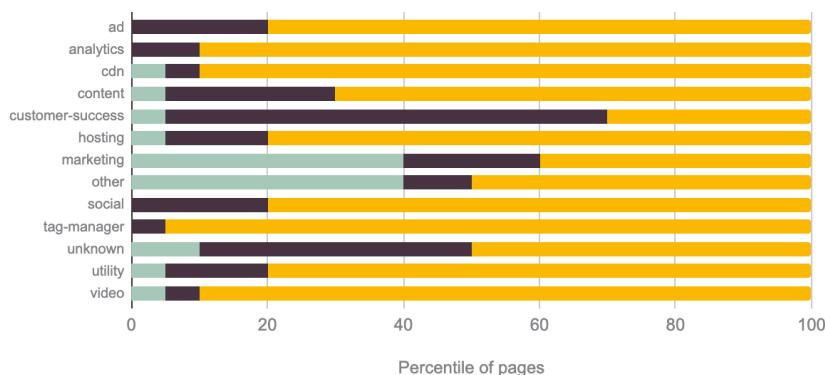



Figure 22.9. The fraction of known third-party HTTP/2 or gQUIC requests by category per website.

Server support

Browser auto-update mechanisms are a driving factor for client-side adoption of new web standards. It's estimated that over 97% of global users support HTTP/2, up slightly from 95% measured last year.

Unfortunately, the upgrade path for servers is more difficult, especially with the requirement to support TLS. For mobile and desktop, we can see from Figure 22.10, that the majority of HTTP/2 sites are served by nginx, Cloudflare, and Apache. Almost half of the HTTP/1.1 sites are served by Apache.

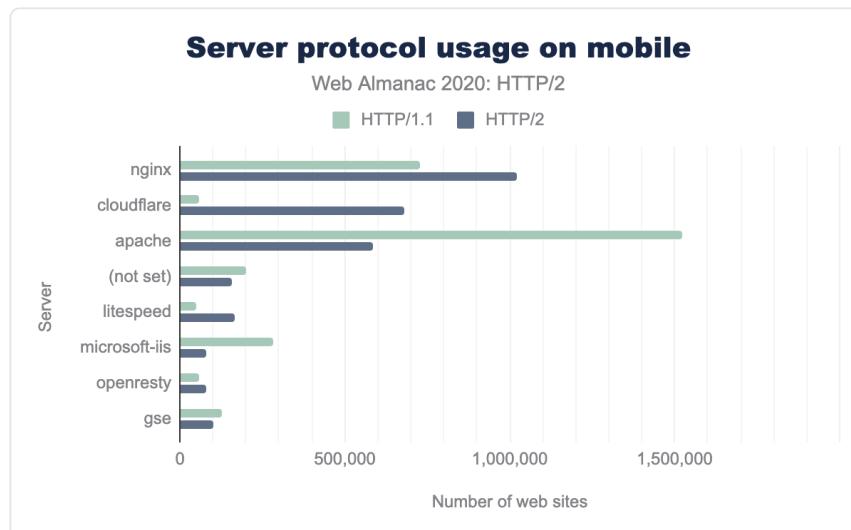


Figure 22.10. Server usage by HTTP protocol on mobile

How has HTTP/2 adoption changed in the last year for each server? Figure 22.11 shows a general HTTP/2 adoption increase of around 10% across all servers since last year. Apache and IIS are still under 25% HTTP/2. This suggests that either new servers tend to be nginx or it is seen as too difficult or not worthwhile to upgrade Apache or IIS to HTTP/2 and/or TLS.

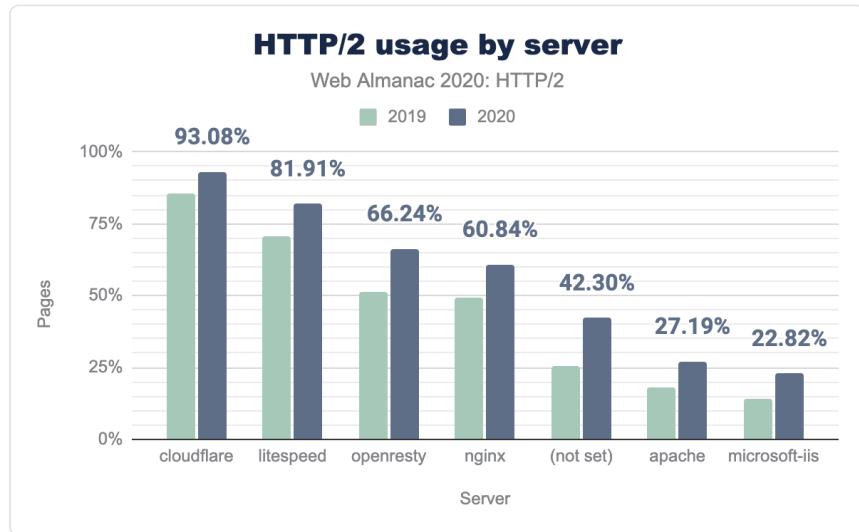


Figure 22.11. Percentage of pages served over HTTP/2 by sever

A long-term recommendation to improve website performance has been to use a CDN. The benefit is a reduction in latency by both serving content and terminating connections closer to the end user. This helps mitigate the rapid evolution in protocol deployment and the additional complexities in tuning servers and operating systems (see the Prioritization section for more details). To utilize the new protocols effectively, using a CDN can be seen as the recommended approach.

CDNs can be classed in two broad categories: those that serve the home page and/or asset subdomains, and those that are mainly used to serve third-party content. Examples of the first category are the larger generic CDNs (such as Cloudflare, Akamai, or Fastly) and the more specific (such as WordPress or Netlify). Looking at the difference in HTTP/2 adoption rates for home pages served with or without a CDN, we see:

- 80% of mobile home pages are served over HTTP/2 if a CDN is used
- 30% of mobile home pages are served over HTTP/2 if a CDN is not used

Figure 22.12 shows the more specific and the modern CDNs serve a higher proportion of traffic over HTTP/2.

HTTP/2 (%)	CDN
100%	Bison Grid, CDNsun, LeaseWeb CDN, NYI FTW, QUIC.cloud, Roast.io, Sirv CDN, Twitter, Zycada Networks
90 - 99%	Automattic, Azion, BitGravity, Facebook, KeyCDN, Microsoft Azure, NGENIX, Netlify, Yahoo, section.io, Airee, BunnyCDN, Cloudflare, GoCache, NetDNA, SFR, Sucuri Firewall
70 - 89%	Amazon CloudFront, BelugaCDN, CDN, CDN77, Estream, Fastly, Highwinds, OVH CDN, Yottaa, Edgecast, Myra Security CDN, StackPath, XLabs Security
20 - 69%	Akamai, Aryaka, Google, Limelight, Rackspace, Incapsula, Level 3, Medianova, OnApp, Singular CDN, Vercel, Cachefly, Cedexis, Reflected Networks, Universal CDN, Yunjiasu, CDNetworks
< 20%	Rocket CDN, BO.LT, ChinaCache, KINX CDN, Zenedge, ChinaNetCenter

Figure 22.12. Percentage of HTTP/2 requests served by the first-party CDNs over mobile.

Types of content in the second category are typically shared resources (JavaScript or font CDNs), advertisements, or analytics. In all these cases, using a CDN will improve the performance and offload for the various SaaS solutions.

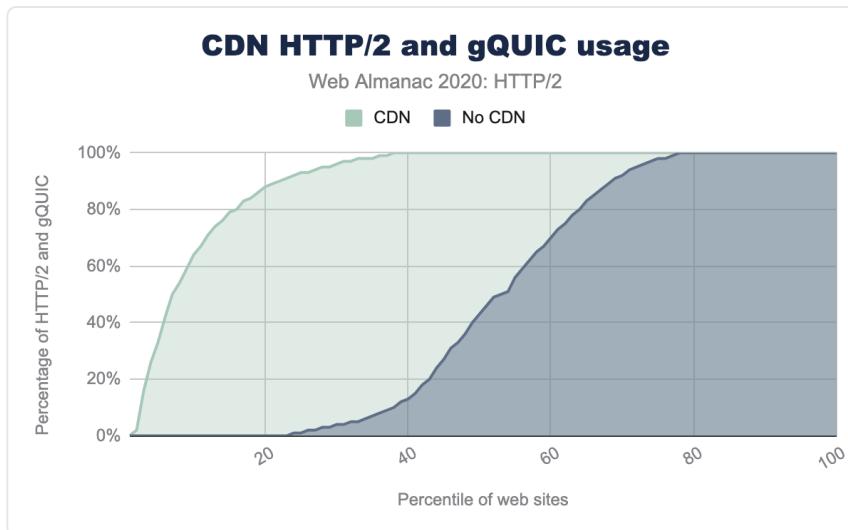


Figure 22.13. Comparison of HTTP/2 and gQUIC usage for websites using a CDN.

In Figure 22.13 we can see the stark difference in HTTP/2 and gQUIC adoption when a website is using a CDN. 70% of pages use HTTP/2 for all third-party requests when a CDN is used. Without a CDN, only 25% of pages use HTTP/2 for all third-party requests.

HTTP/2 impact

Measuring the impact of how a protocol is performing is difficult with the current HTTP Archive approach. It would be really fascinating to be able to quantify the impact of concurrent connections, the effect of packet loss, and different congestion control mechanisms. To really compare performance, each website would have to be crawled over each protocol over different network conditions. What we can do instead is to look into the impact on the number of connections a website uses.

Reducing connections

As discussed earlier, HTTP/1.1 only allows a single request at a time over a TCP connection. Most browsers get around this by allowing six parallel connections per host. The major improvement with HTTP/2 is that multiple requests can be multiplexed over a single TCP connection. This should reduce the total number of connections—and the associated time and resources—required to load a page.

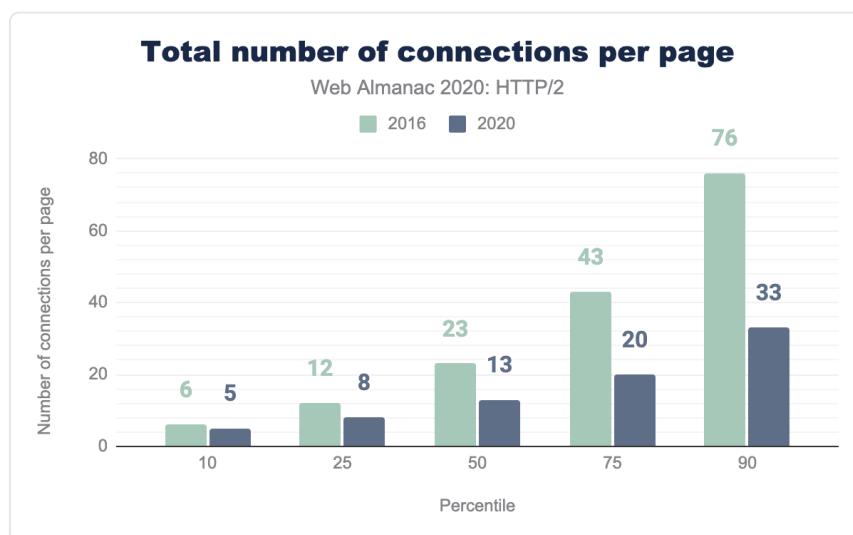


Figure 22.14. Distribution of total number of connections per page

Figure 22.15 shows how the number of TCP connections per page has reduced in 2020 compared with 2016. Half of all websites now use 13 or fewer TCP connections in 2020 compared with 23 connections in 2016; a 44% decrease. In the same time period the median number of requests has only dropped from 74 to 73. The median number of requests per TCP connection has increased from 3.2 to 5.6.

TCP was designed to maintain an average data flow that is both efficient and fair. Imagine a flow control process where each flow both exerts pressure on and is responsive to all other flows, to provide a fair share of the network. In a fair protocol, every TCP session does not crowd out any other session and over time will take $1/N$ of the path capacity.

The majority of websites still open over 15 TCP connections. In HTTP/1.1, the six connections a browser could open to a domain can over time claim six times as much bandwidth as a single HTTP/2 connection. Over low capacity networks, this can slow down the delivery of content from the primary asset domains as the number of contending connections increases and takes bandwidth away from the important requests. This favors websites with a small number of third-party domains.

HTTP/2 does allow for connection reuse across different, but related domains. For a TLS resource, it requires a certificate that is valid for the host in the URI. This can be used to reduce the number of connections required for domains under the control of the site author.

Prioritization

As HTTP/2 responses can be split into many individual frames, and as frames from multiple streams can be multiplexed, the order in which the frames are interleaved and delivered by the server becomes a critical performance consideration. A typical website consists of many different types of resources: the visible content (HTML, CSS, images), the application logic (JavaScript), ads, analytics for tracking site usage, and marketing tracking beacons. With knowledge of how a browser works, an optimal ordering of the resources can be defined that will result in the fastest user experience. The difference between optimal and non-optimal can be significant—as much as a 50% performance improvement or more!

HTTP/2 introduced the concept of prioritization to help the client communicate to the server how it thinks the multiplexing should be done. Every stream is assigned a weight (how much of the available bandwidth the stream should be allocated) and possibly a parent (another stream which should be delivered first). With the flexibility of HTTP/2's prioritization model, it is not altogether surprising that all of the current browser engines implemented different prioritization strategies, none of which are optimal.

There are also problems on the server side, leading to many servers implementing prioritization either poorly or not at all. In the case of HTTP/1.x, tuning the server-side send buffers to be as big as possible has no downside, other than the increase in memory use (trading off memory for CPU), and is an effective way to increase the throughput of a web server. This is not true for HTTP/2, as data in the TCP send buffer cannot be re-prioritized if a request for a new, more important resource comes in. For an HTTP/2 server, the optimal send buffer size is thus the minimum amount of data required to fully utilize the available bandwidth. This allows the

server to respond immediately if a higher-priority request is received.

This problem of large buffers messing with (re-)prioritization also exists in the network, where it goes by the name "bufferbloat". Network equipment would rather buffer packets than drop them when there's a short burst. However, if the server sends more data than the path to the client can consume, these buffers fill to capacity. These bytes already "stored" on the network limit the server's ability to send a higher-priority response earlier, just as a large send buffer does. To minimize the amount of data held in buffers, a recent congestion control algorithm such as BBR should be used.

This test suite maintained by Andy Davies measures and reports how various CDN and cloud hosting services perform. The bad news is that only 9 of the 36 services prioritize correctly. Figure 22.16 shows that for sites using a CDN, around 31.7% do not prioritize correctly. This is up from 26.82% last year, mainly due to the increase in Google CDN usage. Rather than relying on the browser-sent priorities, there are some servers that implement a server side prioritization scheme instead, improving upon the browser's hints with additional logic.

CDN	Prioritize correctly	Desktop	Mobile
Not using CDN	Unknown	59.47%	60.85%
Cloudflare	Pass	22.03%	21.32%
Google	Fail	8.26%	8.94%
Amazon CloudFront	Fail	2.64%	2.27%
Fastly	Pass	2.34%	1.78%
Akamai	Pass	1.31%	1.19%
Automattic	Pass	0.93%	1.05%
Sucuri Firewall	Fail	0.77%	0.63%
Incapsula	Fail	0.42%	0.34%
Netlify	Fail	0.27%	0.20%

Figure 22.15. HTTP/2 prioritization support in common CDNs.

For non-CDN usage, we expect the number of servers that correctly apply HTTP/2 prioritization to be considerably smaller. For example, NodeJS's HTTP/2 implementation does not support prioritization.

Goodbye server push?

Server push was one of the additional features of HTTP/2 that caused some confusion and complexity to implement in practice. Push seeks to avoid waiting for a browser/client to download a HTML page, parse that page, and only then discover that it requires additional resources (such as a stylesheet), which in turn have to be fetched and parsed to discover even more dependencies (such as fonts). All that work and round trips takes time. With server push, in theory, the server can just send multiple responses at once, avoiding the extra round trips.

Unfortunately, with TCP congestion control in play, the data transfer starts off so slowly that not all the assets can be pushed until multiple round trips have increased the transfer rate sufficiently. There are also implementation differences between browsers as the client processing model had not been fully agreed. For example, each browser has a different implementation of a *push cache*.

Another issue is that the server is not aware of resources the browser has already cached. When a server tries to push something that is unwanted, the client can send a `RST_STREAM` frame, but by the time this has happened, the server may well have already sent all the data. This wastes bandwidth and the server has lost the opportunity of immediately sending something that the browser actually did require. There were proposals to allow clients to inform the server of their cache status, but these suffered from privacy concerns.

As can be seen from the Figure 20.17 below, a very small percentage of sites use server push.

Client	HTTP/2 pages	HTTP/2 (%)	gQUIC pages	gQUIC (%)
Desktop	44,257	0.85%	204	0.04%
Mobile	62,849	1.06%	326	0.06%

Figure 22.16. Pages using HTTP/2 or gQUIC server push.

Looking further at the distributions for pushed assets in Figures 22.18 and 22.19, half of the sites push 4 or fewer resources with a total size of 140 KB on desktop and 3 or fewer resources with a size of 184 KB on mobile. For gQUIC, desktop is 7 or fewer and mobile 2. The worst offending page pushes 41 assets over gQUIC on desktop.

Percentile	HTTP/2	Size (KB)	gQUIC	Size (KB)
10	1	15.48	1	0.06
25	1	36.34	1	0.06
50	3	183.83	2	24.06
75	10	225.41	5	204.65
90	12	351.05	18	453.57

Figure 22.17. Distribution of pushed assets on desktop.

Percentile	HTTP/2	Size (KB)	gQUIC	Size (KB)
10	1	15.48	1	0.06
25	1	36.34	1	0.06
50	3	183.83	2	24.06
75	10	225.41	5	204.65
90	12	351.05	18	453.57

Figure 22.18. Distribution of pushed assets on mobile.

Looking at the frequency of push by content type in Figure 22.20, we see 90% of pages push scripts and 56% push CSS. This makes sense, as these can be small files typically on the critical path to render a page.

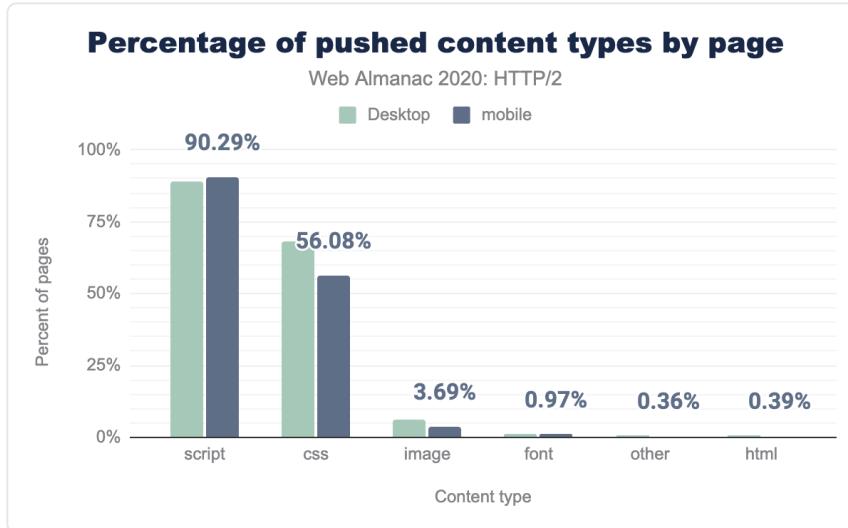


Figure 22.19. Percentage of pages pushing specific content types

Given the low adoption, and after measuring how few of the pushed resources are actually useful (that is, they match a request that is not already cached), Google has announced the intent to remove push support from Chrome for both HTTP/2 and gQUIC. Chrome has also not implemented push for HTTP/3.

Despite all these problems, there are circumstances where server push can provide an improvement. The ideal use case is to be able to send a push promise much earlier than the HTML response itself. A scenario where this can benefit is when a CDN is in use. The "dead time" between the CDN receiving the request and receiving a response from the origin can be used intelligently to warm up the TCP connection and push assets already cached at the CDN.

There was however no standardized method for how to signal to a CDN edge server that an asset should be pushed. Implementations instead reused the preload HTTP link header to indicate this. This simple approach appears elegant, but it does not utilize the dead time before the HTML is generated unless the headers are sent before the actual content is ready. It triggers the edge to push resources as the HTML is received at the edge, which will contend with the delivery of the HTML.

An alternative proposal being tested is RFC 8297, which defines an informative [103 \(Early Hints\)](#) response. This permits headers to be sent immediately, without having to wait for the server to generate the full response headers. This can be used by an origin to suggest pushed resources to a CDN, or by a CDN to alert the client to resources that need to be fetched. However, at present, support for this from both a client and server perspective is very low, but

growing.

Getting to a better protocol

Let's say a client and server support both HTTP/1.1 and HTTP/2. How do they choose which one to use? The most common method is TLS Application Layer Protocol Negotiation (ALPN), in which clients send a list of protocols they support to the server, which picks the one it prefers to use for that connection. Because the browser needs to negotiate the TLS parameters as part of setting up an HTTPS connection, it can also negotiate the HTTP version at the same time. Since both HTTP/2 and HTTP/1.1 can be served from the same TCP port (443), browsers don't need to make this selection before opening a connection.

This doesn't work if the protocols aren't on the same port, use a different transport protocol (TCP versus QUIC), or if you're not using TLS. For those scenarios, you start with whatever is available on the first port you connect to, then discover other options. HTTP defines two mechanisms to change protocols for an origin after connecting: `Upgrade` and `Alt-Svc`.

`Upgrade`

The `Upgrade` header has been part of HTTP for a long time. In HTTP/1.x, `Upgrade` allows a client to make a request using one protocol, but indicate its support for another protocol (like HTTP/2). If the server also supports the offered protocol, it responds with a status 101 (`Switching Protocols`) and proceeds to answer the request in the new protocol. If not, the server answers the request in HTTP/1.x. Servers can advertise their support of a different protocol using an `Upgrade` header on a response.

The most common application of `Upgrade` is WebSockets. HTTP/2 also defines an `Upgrade` path, for use with its unencrypted cleartext mode. There is no support for this capability in web browsers, however. Therefore, it's not surprising that less than 3% of cleartext HTTP/1.1 requests in our dataset received an `Upgrade` header in the response. A very small number of requests using TLS (0.0011% of HTTP/2, 0.064% of HTTP/1.1) also received `Upgrade` headers in response; these are likely cleartext HTTP/1.1 servers behind intermediaries which speak HTTP/2 and/or terminate TLS, but don't properly remove `Upgrade` headers.

Alternative Services

Alternative Services (`Alt-Svc`) enables an HTTP origin to indicate other endpoints which serve the same content, possibly over different protocols. Although uncommon, HTTP/2 might be located at a different port or different host from a site's HTTP/1.1 service. More importantly,

since HTTP/3 uses QUIC (hence UDP) where prior versions of HTTP use TCP, HTTP/3 will always be at a different endpoint from the HTTP/1.x and HTTP/2 service.

When using `Alt-Svc`, a client makes requests to the origin as normal. However, if the server includes a header or sends a frame containing a list of alternatives, the client can make a new connection to the other endpoint and use it for future requests to that origin.

Unsurprisingly, `Alt-Svc` usage is found almost entirely from services using advanced HTTP versions: 12.0% of HTTP/2 requests and 60.1% of gQUIC requests received an `Alt-Svc` header in response, as compared to 0.055% of HTTP/1.x requests. Note that our methodology here only captures `Alt-Svc` headers, not `ALTSVC` frames in HTTP/2, so reality might be slightly understated.

While `Alt-Svc` can point to an entirely different host, support for this capability varies among browsers. Only 4.71% of `Alt-Svc` headers advertised an endpoint on a different hostname; these were almost universally (99.5%) advertising gQUIC and HTTP/3 support on Google Ads. Google Chrome ignores cross-host `Alt-Svc` advertisements for HTTP/2, so many of the other instances would have been ignored.

Given the rarity of support for cross-host HTTP/2, it's not surprising that there were virtually no (0.007%) advertisements for HTTP/2 endpoints using `Alt-Svc`. `Alt-Svc` was typically used to indicate support for HTTP/3 (74.6% of `Alt-Svc` headers) or gQUIC (38.7% of `Alt-Svc` headers).

Looking toward the future: HTTP/3

HTTP/2 is a powerful protocol, which has found considerable adoption in just a few years. However, HTTP/3 over QUIC is already peeking around the corner! Over four years in the making, this next version of HTTP is almost standardized at the IETF (expected in the first half of 2021). At this time, there are already many QUIC and HTTP/3 implementations available, both for servers and browsers. While these are relatively mature, they can still be said to be in an experimental state.

This is reflected by the usage numbers in the HTTP Archive data, where no HTTP/3 requests were identified at all. This might seem a bit strange, since Cloudflare has had experimental HTTP/3 support for some time, as have Google and Facebook. This is mainly because Chrome hadn't enabled the protocol by default when the data was collected.

However, even the numbers for the older gQUIC are relatively modest, accounting for less than 2% of requests overall. This is expected, since gQUIC was mostly deployed by Google and Akamai; other parties were waiting for IETF QUIC. As such, gQUIC is expected to be replaced

entirely by HTTP/3 soon.

1.72%

Figure 22.20. The percentage of requests that use gQUIC on desktop and mobile

It's important to note that this low adoption only reflects gQUIC and HTTP/3 usage for loading Web pages. For several years already, Facebook has had a much more extensive deployment of IETF QUIC and HTTP/3 for loading data inside of its native applications. QUIC and HTTP/3 already make up over 75% of their total internet traffic. It is clear that HTTP/3 is only just getting started!

Now you might wonder: if not everyone is already using HTTP/2, why would we need HTTP/3 so soon? What benefits can we expect in practice? Let's take a closer look at its internal mechanisms.

QUIC and HTTP/3

Past attempts to deploy new transport protocols on the internet have proven difficult, for example Stream Control Transmission Protocol (SCTP). QUIC is a new transport protocol that runs on top of UDP. It provides similar features to TCP, such as reliable in-order delivery and congestion control to prevent flooding the network.

As discussed in the HTTP/1.0 to HTTP/2 section, HTTP/2 *multiplexes* multiple different streams on top of one connection. TCP itself is woefully unaware of this fact, leading to inefficiencies or performance impact when packet loss or delays occur. More details on this problem, known as *head-of-line blocking* (HOL blocking), can be found here.

QUIC solves the HOL blocking problem by bringing HTTP/2's streams down into the transport layer and performing per-stream loss detection and retransmission. So then we just put HTTP/2 over QUIC, right? Well, we've already mentioned some of the difficulties arising from having flow control in TCP and HTTP/2. Running two separate and competing streaming systems on top of each other would be an additional problem. Furthermore, because the QUIC streams are independent, it would mess with the strict ordering requirements HTTP/2 requires for several of its systems.

In the end, it was deemed easier to define a new HTTP version that uses QUIC directly and thus, HTTP/3 was born. Its high-level features are very similar to those we know from HTTP/2, but internal implementation mechanisms are quite different. HPACK header compression is

replaced with QPACK, which allows manual tuning of the compression efficiency versus HOL blocking risk tradeoff, and the prioritization system is being replaced by a simpler one. The latter could also be back-ported to HTTP/2, possibly helping resolve the prioritization issues discussed earlier in this article. HTTP/3 continues to provide a server push mechanism, but Chrome, for example, does not implement it.

Another benefit of QUIC is that it is able to migrate connections and keep them alive even when the underlying network changes. A typical example is the so-called "parking lot problem". Imagine your smartphone is connected to the workplace Wi-Fi network and you've just started downloading a large file. As you leave Wi-Fi range, your phone automatically switches to the fancy new 5G cellular network. With plain old TCP, the connection would break and cause an interruption. But QUIC is smarter; it uses a *connection ID*, which is more robust to network changes, and provides an active *connection migration* feature for clients to switch without interruption.

Finally, TLS is already used to protect HTTP/1.1 and HTTP/2. QUIC, however, has a deep integration of TLS 1.3, protecting both HTTP/3 data and QUIC packet metadata, such as packet numbers. Using TLS in this way improves end-user privacy and security and makes continued protocol evolution easier. Combining the transport and cryptographic handshakes means that connection setup takes just a single RTT, compared to TCP's minimum of two and worst case of four. In some cases, QUIC can even go one step further and send HTTP data along with its very first message, which is called 0-RTT. These fast connection setup times are expected to really help HTTP/3 outperform HTTP/2.

So, will HTTP/3 help?

On the surface, HTTP/3 is really not all that different from HTTP/2. It doesn't add any major features, but mainly changes how the existing ones work under the surface. The real improvements come from QUIC, which offers faster connection setups, increased robustness, and resilience to packet loss. As such, HTTP/3 is expected to do better than HTTP/2 on worse networks, while offering very similar performance on faster systems. However, that is if the web community can get HTTP/3 working, which can be challenging in practice.

Deploying and discovering HTTP/3

Since QUIC and HTTP/3 run over UDP, things aren't as simple as with HTTP/1.1 or HTTP/2. Typically, an HTTP/3 client has to first discover that QUIC is available at the server. The recommended method for this is HTTP Alternative Services . On its first visit to a website, a client connects to a server using TCP. It then discovers via `Alt-Svc` that HTTP/3 is available, and can set up a new QUIC connection. The `Alt-Svc` entry can be cached, allowing subsequent visits to avoid the TCP step, but the entry will eventually become stale and need

revalidation. This likely will have to be done for each domain separately, which will probably lead to most page loads using a mix of HTTP/1.1, HTTP/2, and HTTP/3.

However, even if it is known that a server supports QUIC and HTTP/3, the network in between might block it. UDP traffic is commonly used in DDoS attacks and blocked by default in for example many company networks. While exceptions could be made for QUIC, its encryption makes it difficult for firewalls to assess the traffic. There are potential solutions to these issues, but in the meantime it is expected that QUIC is most likely to succeed on well-known ports like 443. And it is entirely possible that it is blocked QUIC altogether. In practice, clients will likely use sophisticated mechanisms to fall back to TCP if QUIC fails. One option is to "race" both a TCP and QUIC connection and use the one that completes first.

There is ongoing work to define ways to discover HTTP/3 without needing the TCP step. This should be considered an optimization, though, as the UDP blocking issues are likely to mean that TCP-based HTTP sticks around. The HTTPS DNS record is similar to HTTP Alternative Services and some CDNs are already experimenting with these records. In the long run, when most servers offer HTTP/3, browsers might switch to attempting that by default; that will take a long time.

TLS version	HTTP/1.x desktop	HTTP/1.x mobile	HTTP/2 desktop	HTTP/2 mobile
unknown	4.06%	4.03%	5.05%	7.28%
TLS 1.2	26.56%	24.75%	23.12%	23.14%
TLS 1.3	5.25%	5.11%	35.78%	35.54%

Figure 22.21. TLS adoption by HTTP version.

QUIC is dependent on TLS 1.3, which is used for around 41% of requests, as shown in Figure 22.21. That leaves 59% of requests that will need to update their TLS stack to support HTTP/3.

Is HTTP/3 ready yet?

So, when can we start using HTTP/3 and QUIC for real? Not quite yet, but hopefully soon. There is a large number of mature open source implementations and the major browsers have experimental support. However, most of the typical servers have suffered some delays: nginx is a bit behind other stacks, Apache hasn't announced official support, and NodeJS relies on OpenSSL, which won't add QUIC support anytime soon. Even so, we expect to see HTTP/3 and QUIC deployments rise throughout 2021.

HTTP/3 and QUIC are highly advanced protocols with powerful performance and security features, such as a new HTTP prioritization system, HOL blocking removal, and 0-RTT connection establishment. This sophistication also makes them difficult to deploy and use correctly, as has turned out to be the case for HTTP/2. We predict that early deployments will mainly be done via the early adoption of CDNs such as Cloudflare, Fastly, and Akamai. This will probably mean that a large part of HTTP/2 traffic can and will be upgraded to HTTP/3 automatically in 2021, giving the new protocol a large traffic share almost out of the box. When and if smaller deployments will follow suit is more difficult to answer. Most likely, we will continue to see a healthy mix of HTTP/3, HTTP/2, and even HTTP/1.1 on the web for years to come.

Conclusion

This year, HTTP/2 has become the dominant protocol, serving 64% of all requests, having grown by 10 percentage points during the year. Home pages have seen a 13% increase in HTTP/2 adoption, leading to an even split of pages served over HTTP/1.1 and HTTP/2. Using a CDN to serve your home page pushes HTTP/2 adoption up to 80%, compared with 30% for non-CDN usage. There remain some older servers, Apache and IIS, that are proving resistant to upgrading to HTTP/2 and TLS. A big success has been the decrease in website connection usage due to HTTP/2 connection multiplexing. The median number of connections in 2016 was 23 compared to 13 in 2020.

HTTP/2 prioritization and server push have turned out to be way more complex to deploy at large. Under certain implementations they show clear performance benefits. There is, however, a significant barrier to deploying and tuning existing servers to use these features effectively. There are still a large proportion of CDNs who do not support prioritization effectively. There have also been issues with consistent browser support.

HTTP/3 is just around the corner. It will be fascinating to follow the adoption rate, see how discovery mechanisms evolve, and find out which new features will be deployed successfully. We expect next year's Web Almanac to see some interesting new data.

Authors



Andrew Galloni

@dot_js dotjs

Andrew works at Cloudflare⁶⁶ helping to make the web faster and more secure. He spends his time deploying, measuring and improving new protocols and asset delivery to improve end-user website performance.



Robin Marx

@programmingart rmarx

Robin is a web protocol and performance researcher at Hasselt University, Belgium⁶⁷. He has been working on getting QUIC and HTTP/3 ready to use by creating tools like qlog and qvis⁶⁸.

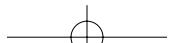
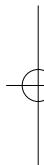
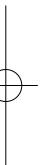


Mike Bishop

MikeBishop

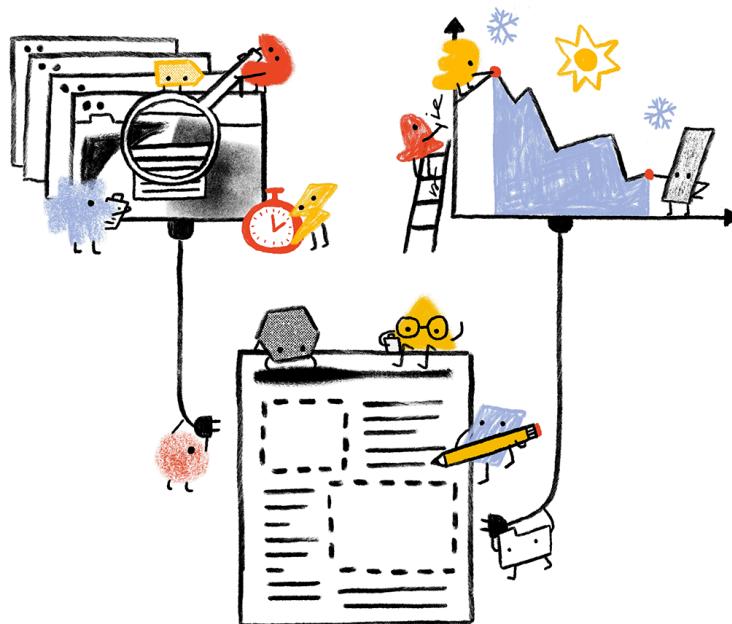
Editor of HTTP/3 with the QUIC Working Group⁶⁹. Architect in Akamai⁷⁰'s Foundry group.

66. <https://www.cloudflare.com/>
67. <https://www.uhasselt.be/edm>
68. <https://github.com/quiclog>
69. <https://quicwg.org/>
70. <https://www.akamai.com/>



Appendix A

Methodology



Overview

The Web Almanac is a project organized by HTTP Archive⁷¹. HTTP Archive was started in 2010 by Steve Souders with the mission to track how the web is built. It evaluates the composition of millions of web pages on a monthly basis and makes its terabytes of metadata available for analysis on BigQuery⁷².

The Web Almanac's mission is to become an annual repository of public knowledge about the

71. <https://httparchive.org>
72. <https://httparchive.org/faq#how-do-i-use-bigquery-to-write-custom-queries-over-the-data>

state of the web. Our goal is to make the data warehouse of HTTP Archive even more accessible to the web community by having subject matter experts provide contextualized insights.

The 2020 edition of the Web Almanac is broken into four parts: content, experience, publishing, and distribution. Within each part, several chapters explore their overarching theme from different angles. For example, Part II explores different angles of the user experience in the Performance, Security, and Accessibility chapters, among others.

About the dataset

The HTTP Archive dataset is continuously updating with new data monthly. For the 2020 edition of the Web Almanac, unless otherwise noted in the chapter, all metrics were sourced from the August 2020 crawl. These results are publicly queryable⁷³ on BigQuery in tables prefixed with `2020_08_01`.

All of the metrics presented in the Web Almanac are publicly reproducible using the dataset on BigQuery. You can browse the queries used by all chapters in our GitHub repository⁷⁴.

Please note that some of these queries are quite large and can be expensive⁷⁵ to run yourself, as BigQuery is billed by the terabyte. For help controlling your spending, refer to Tim Kadlec's post Using BigQuery Without Breaking the Bank⁷⁶.

For example, to understand the median number of bytes of JavaScript per desktop and mobile page, see `01_01b.sql`⁷⁷:

```
#standardSQL
# 01_01b: Distribution of JS bytes by client
SELECT
    percentile,
    _TABLE_SUFFIX AS client,
    APPROX_QUANTILES(ROUND(bytesJs / 1024, 2),
    1000)[OFFSET(percentile * 10)] AS js_kbytes
```

73. https://github.com/HTTPArchive/httparchive.org/blob/master/docs/gettingstarted_bigquery.md

74. <https://github.com/HTTPArchive/almanac.httparchive.org/tree/main/sql/2020>

75. <https://cloud.google.com/bigquery/pricing>

76. <https://timkadlec.com/remembers/2019-12-10-using-bigquery-without-breaking-the-bank/>

77. https://github.com/HTTPArchive/almanac.httparchive.org/blob/main/sql/2019/01_JavaScript/01_01b.sql

```

FROM
`httparchive.summary_pages.2019_07_01_*`,
UNNEST([10, 25, 50, 75, 90]) AS percentile
GROUP BY
percentile,
client
ORDER BY
percentile,
client

```

Results for each metric are publicly viewable in chapter-specific spreadsheets, for example JavaScript results⁷⁸. Scroll to the bottom of each chapter for links to their queries, results, and comments from readers.

Websites

There are 7,546,709 websites in the dataset. Among those, 6,347,919 are mobile websites and 5,593,642 are desktop websites. Most websites are included in both the mobile and desktop subsets.

HTTP Archive sources the URLs for its websites from the Chrome UX Report. The Chrome UX Report is a public dataset from Google that aggregates user experiences across millions of websites actively visited by Chrome users. This gives us a list of websites that are up-to-date and a reflection of real-world web usage. The Chrome UX Report dataset includes a form factor dimension, which we use to get all of the websites accessed by desktop or mobile users.

The August 2020 HTTP Archive crawl used by the Web Almanac used the most recently available Chrome UX Report release for its list of websites. The 202006 dataset was released on July 14, 2020 and captures websites visited by Chrome users during the month of June.

There was a 20-30% growth in the number of websites analyzed compared to those in the 2019 Web Almanac. This increase has been analyzed by Paul Calvano in his Growth of the Web in 2020⁷⁹ post.

78. https://docs.google.com/spreadsheets/d/1kBtqjETN_V9UjKqK_EFmFjRexJnQOmLLr-l2Tkotvic/edit?usp=sharing

79. <https://paulcalvano.com/2020-09-29-growth-of-the-web-in-2020/>

Due to resource limitations, the HTTP Archive can only test one page from each website in the Chrome UX report. To reconcile this, only the home pages are included. Be aware that this will introduce some bias into the results because a home page is not necessarily representative of the entire website.

HTTP Archive is also considered a lab testing tool, meaning it tests websites from a datacenter and does not collect data from real-world user experiences. All pages are tested with an empty cache in a logged out state, which may not reflect how real users would access them.

Metrics

HTTP Archive collects thousands of metrics about how the web is built. It includes basic metrics like the number of bytes per page, whether the page was loaded over HTTPS, and individual request and response headers. The majority of these metrics are provided by WebPageTest, which acts as the test runner for each website.

Other testing tools are used to provide more advanced metrics about the page. For example, Lighthouse is used to run audits against the page to analyze its quality in areas like accessibility and SEO. The Tools section below goes into each of these tools in more detail.

To work around some of the inherent limitations of a lab dataset, the Web Almanac also makes use of the Chrome UX Report for metrics on user experiences, especially in the area of web performance.

Some metrics are completely out of reach. For example, we don't necessarily have the ability to detect the tools used to build a website. If a website is built using create-react-app, we could tell that it uses the React framework, but not necessarily that a particular build tool is used. Unless these tools leave detectable fingerprints in the website's code, we're unable to measure their usage.

Other metrics may not necessarily be impossible to measure but are challenging or unreliable. For example, aspects of web design are inherently visual and may be difficult to quantify, like whether a page has an intrusive modal dialog.

Tools

The Web Almanac is made possible with the help of the following open source tools.

WebPageTest

WebPageTest⁸⁰ is a prominent web performance testing tool and the backbone of HTTP Archive. We use a private instance⁸¹ of WebPageTest with private test agents, which are the actual browsers that test each web page. Desktop and mobile websites are tested under different configurations:

Config	Desktop	Mobile
Device	Linux VM	Emulated Moto G4
User Agent	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/84.0.4147.105 Safari/537.36 PTST/200805.230825	Mozilla/5.0 (Linux; Android 6.0.1; Moto G (4) Build/MPJ24.139-64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.146 Mobile Safari/537.36 PTST/200815.130813
Location	Redwood City, California, USA The Dalles, Oregon, USA	Redwood City, California, USA The Dalles, Oregon, USA
Connection	Cable (5/1 Mbps 28ms RTT)	3G (1.600/0.768 Mbps 300ms RTT)
Viewport	1024 x 768px	512 x 360px

Desktop websites are run from within a desktop Chrome environment on a Linux VM. The network speed is equivalent to a cable connection.

Mobile websites are run from within a mobile Chrome environment on an emulated Moto G4 device with a network speed equivalent to a 3G connection. Note that the emulated mobile User Agent self-identifies as Chrome 65 but is actually Chrome 84 under the hood.

There are two locations from which tests are run: California and Oregon USA. HTTP Archive maintains its own test agent hardware located in the Internet Archive⁸² datacenter in California. Additional test agents in Google Cloud Platform⁸³'s us-west-1 location in Oregon are added as needed.

HTTP Archive's private instance of WebPageTest is kept in sync with the latest public version and augmented with custom metrics⁸⁴. These are snippets of JavaScript that are evaluated on each website at the end of the test. Thanks to the contributions⁸⁵ of many data analysts,

80. <https://www.webpagetest.org/>
 81. <https://github.com/WPO-Foundation/webpagetest-docs/blob/master/user/Private%20Instances/README.md>
 82. <https://archive.org>
 83. <https://cloud.google.com/compute/docs/regions-zones/#locations>
 84. https://github.com/HTTPArchive/legacy.httparchive.org/tree/master/custom_metrics
 85. https://github.com/HTTPArchive/legacy.httparchive.org/commits/master/custom_metrics

especially the herculean efforts⁸⁶ of Tony McCreath, the 2020 edition of the Web Almanac greatly expanded the capabilities of HTTP Archive's test infrastructure with over 3,000 lines of new code.

The results of each test are made available as a HAR file⁸⁷, a JSON-formatted archive file containing metadata about the web page.

Lighthouse

Lighthouse⁸⁸ is an automated website quality assurance tool built by Google. It audits web pages to make sure they don't include user experience antipatterns like unoptimized images and inaccessible content.

HTTP Archive runs the latest version of Lighthouse for all of its mobile web pages – desktop pages are not included because of limited resources. As of the August 2020 crawl, HTTP Archive used the 6.2.0⁸⁹ version of Lighthouse.

Lighthouse is run as its own distinct test from within WebPageTest, but it has its own configuration profile:

Config	Value
CPU slowdown	1x/4x
Download throughput	1.6 Mbps
Upload throughput	0.768 Mbps
RTT	150 ms

For more information about Lighthouse and the audits available in HTTP Archive, refer to the Lighthouse developer documentation⁹⁰.

Wappalyzer

Wappalyzer⁹¹ is a tool for detecting technologies used by web pages. There are 64 categories⁹² of technologies tested, ranging from JavaScript frameworks, to CMS platforms, and even

86. <https://github.com/HTTPArchive/legacy.httparchive.org/pulls?q=is%3Apr+author%3ATiggerito+sort%3Acreated-asc>
 87. https://en.wikipedia.org/wiki/HAR_file_format
 88. <https://developers.google.com/web/tools/lighthouse/>
 89. <https://github.com/GoogleChrome/lighthouse/releases/tag/v6.2.0>
 90. <https://developers.google.com/web/tools/lighthouse/>
 91. <https://www.wappalyzer.com/>
 92. <https://www.wappalyzer.com/technologies>

cryptocurrency miners. There are over 1,400 supported technologies.

HTTP Archive runs the latest version of Wappalyzer for all web pages. As of August 2020 the Web Almanac used the 6.2.0 version⁹³ of Wappalyzer.

Wappalyzer powers many chapters that analyze the popularity of developer tools like WordPress, Bootstrap, and jQuery. For example, the Ecommerce and CMS chapters rely heavily on the respective Ecommerce⁹⁴ and CMS⁹⁵ categories of technologies detected by Wappalyzer.

All detection tools, including Wappalyzer, have their limitations. The validity of their results will always depend on how accurate their detection mechanisms are. The Web Almanac will add a note in every chapter where Wappalyzer is used but its analysis may not be accurate due to a specific reason.

Chrome UX Report

The Chrome UX Report⁹⁶ is a public dataset of real-world Chrome user experiences.

Experiences are grouped by websites' origin, for example <https://www.example.com>. The dataset includes distributions of UX metrics like paint, load, interaction, and layout stability. In addition to grouping by month, experiences may also be sliced by dimensions like country-level geography, form factor (desktop, phone, tablet), and effective connection type (4G, 3G, etc.).

For Web Almanac metrics that reference real-world user experience data from the Chrome UX Report, the August 2020 dataset (202008) is used.

You can learn more about the dataset in the Using the Chrome UX Report on BigQuery⁹⁷ guide on web.dev⁹⁸.

Third Party Web

Third Party Web⁹⁹ is a research project by Patrick Hulce, author of the 2019 Third Parties chapter, that uses HTTP Archive and Lighthouse data to identify and analyze the impact of third party resources on the web.

Domains are considered to be a third party provider if they appear on at least 50 unique pages. The project also groups providers by their respective services in categories like ads, analytics, and social.

93. <https://github.com/AliasIO/Wappalyzer/releases/tag/v6.2.0>
 94. <https://www.wappalyzer.com/categories/ecommerce>
 95. <https://www.wappalyzer.com/categories/cms>
 96. <https://developers.google.com/web/tools/chrome-user-experience-report>
 97. <https://web.dev/chrome-ux-report-bigquery>
 98. <https://web.dev/>
 99. <https://www.thirdpartyweb.today/>

Several chapters in the Web Almanac use the domains and categories from this dataset to understand the impact of third parties.

Rework CSS

Rework CSS¹⁰⁰ is a JavaScript-based CSS parser. It takes entire stylesheets and produces a JSON-encoded object distinguishing each individual style rule, selector, directive, and value.

This special purpose tool significantly improved the accuracy of many of the metrics in the CSS chapter. CSS in all external stylesheets and inline style blocks for each page were parsed and queried to make the analysis possible. See this thread¹⁰¹ for more information about how it was integrated with the HTTP Archive dataset on BigQuery.

Rework Utils

This year's CSS chapter led by Lea Verou took a significantly more detailed look at the state of CSS, spread over 100+ queries¹⁰². For perspective, that's 2.5x more queries than in 2019. To make this scale of analysis feasible, Lea open sourced the Rework Utils¹⁰³. These utilities take the JSON data from Rework to the next level by providing helpful scripts to more easily extract CSS insights. Most of the stats you see in the CSS chapter are powered by these scripts.

Parsel

Parsel¹⁰⁴ is a CSS selector parser and specificity calculator, originally written by CSS chapter lead Lea Verou and open sourced as a separate library. It is used extensively in all CSS metrics that relate to selectors and specificity.

Analytical process

The Web Almanac took about a year to plan and execute with the coordination of more than a hundred contributors from the web community. This section describes why we chose the chapters you see in the Web Almanac, how their metrics were queried, and how they were interpreted.

100. <https://github.com/reworkcss/css>
101. <https://discuss.httparchive.org/t/analyzing-style-sheets-with-a-js-based-parser/1683>
102. https://github.com/HTTPArchive/almacen.httparchive.org/tree/main/sql/2020/01_CSS
103. <https://github.com/LeaVerou/rework-utils>
104. <https://projects.verou.me/parsel/>

Planning

The 2020 Web Almanac kicked off in June 2020¹⁰⁵, later than the 2019 timeline due to the unrest related to COVID-19 and the social justice protests. These and other events of 2020 were an undercurrent throughout the entire production process and put a lot of additional strain on our contributors beyond the stresses of a fast-paced project like this.

As we stated in last year's Methodology¹⁰⁶:

One explicit goal for future editions of the Web Almanac is to encourage even more inclusion of underrepresented and heterogeneous voices as authors and peer reviewers.

To that end, this year we've made systematic changes to the way that we seek and select authors:

- Previous authors were specifically discouraged from writing again to make room for different perspectives.
- Everyone endorsing 2020 authors were asked to be especially conscious not to nominate people who all look or think alike.
- Many 2019 authors were Google employees and this year we tried to get a greater balance of perspectives from the broader web community. We believe that the voices in the Web Almanac should be a reflection of the community itself, and not skewed towards any specific company to avoid creating echo chambers.
- The project leads reviewed all of the author nominations and made an effort to select authors who will bring new perspectives and amplify the voices of underrepresented groups in the community.

We hope to iterate on this process in the future to ensure that the Web Almanac is a more diverse and inclusive project with contributors from all backgrounds.

Finally, in July 2020, after rounds of brainstorming and nominations, 22 chapters were solidified and we formed content teams for each chapter tasked with writing, reviewing, and analysis.

Analysis

In July and August 2020, with the stable list of metrics and chapters, data analysts triaged the metrics for feasibility. In some cases, custom metrics¹⁰⁷ were created to fill gaps in our analytic

105. https://twitter.com/rick_visconti/status/1273135952848977920

106. <https://almanac.httparchive.org/en/2019/methodology/#brainstorming>

107. https://github.com/HTTPArchive/legacy.httparchive.org/tree/master/custom_metrics

capabilities.

Throughout August 2020, the HTTP Archive data pipeline crawled several million websites, gathering the metadata to be used in the Web Almanac.

The data analysts began writing queries to extract the results for each metric. In total, hundreds of queries were written by hand! You can browse all of the queries by year and chapter in our open source query repository¹⁰⁸ on GitHub.

Interpretation

Authors worked with analysts to correctly interpret the results and draw appropriate conclusions. As authors wrote their respective chapters, they drew from these statistics to support their framing of the state of the web. Peer reviewers worked with authors to ensure the technical correctness of their analysis.

To make the results more easily understandable to readers, web developers and analysts created data visualizations to embed in the chapter. Some visualizations are simplified to make the points more clearly. For example, rather than showing a full distribution, only a handful of percentiles are shown. Unless otherwise noted, all distributions are summarized using percentiles, especially medians (the 50th percentile), and not averages.

Finally, editors revised the chapters to fix simple grammatical errors and ensure consistency across the reading experience.

Looking ahead

The 2020 edition of the Web Almanac is the second in what we hope to continue as an annual tradition in the web community of introspection and a commitment to positive change. Getting to this point has been a monumental effort thanks to many dedicated contributors and we hope to leverage as much of this work as possible to make future editions even more streamlined.

If you're interested in contributing to the 2021 edition of the Web Almanac, please fill out our interest form¹⁰⁹. Let's work together to track the state of the web!

108. <https://github.com/HTTPArchive/almanac.httparchive.org/tree/main/sql/2020>
109. <https://forms.gle/VRBFegGAP7d99Bhp7>

Appendix B

Contributors



The Web Almanac has been made possible by the hard work of the web community. 115 people have volunteered countless hours in the planning, research, writing and production phases.

**Abby Tsai**

@AbbyTsai

Analysts, Developers, Translators

**Alex Tait**

@at_fresh_dev

alextait1

<https://togethertech.ca/>

Authors

**Aditya Pandey**

@adityapandey98

adityapandey1998

adityapandey98

Developers

**Alexey Pyltsyn**

lex111

<https://lex111.ru/>

Developers, Editors, Translators

**Adrian Roselli**

@aardrian

aardrian

<https://adrianroselli.com/>

Reviewers

**Aleyda Solis**

@aleyda

aleyda

<https://www.aleydasolis.com/en/>

Authors

**Ahmad Awais**

@MrAhmadAwais

ahmadawais

<https://AhmadAwais.com>

Authors

**Andrew Galloni**

@dot_js

dotjs

Authors

**Alberto Medina**

@AlbMedina

amedina

Reviewers

**Andy Bell**

@hankchizljaw

hankchizljaw

<https://hankchizljaw.com/>

Reviewers

**Alex Denning**

@AlexDenning

alexdenning

<https://getellipsis.com>

Authors

**Antoine Eripret**

antoineeripret

Analysts



Artem Denysov

✉ @denar90_
⌚ denar90
Analysts Reviewers



Catalin Rosu

✉ @catalinred
⌚ catalinred
🌐 https://catalin.red/
Authors, Developers, Reviewers



Barry Pollard

✉ @tunetheweb
⌚ bazzadp
📠 barry-pollard-developer
🌐 https://www.tunetheweb.com
Analysts, Authors, Developers, Editors, Project Leads, Reviewers



Cheng Xi

⌚ chengxicn
Translators



Ben Seymour

⌚ bseymour
🌐 https://benseymour.com
Authors



Chris Lilley

✉ @svgeesus
⌚ svgeesus
🌐 https://svgees.us/
Authors Reviewers



Bharat Agarwal

⌚ bharatagsrwal
🌐 https://iambharat.me
Developers



Christian Liebel

✉ @christianliebel
⌚ christianliebel
🌐 https://christianliebel.com
Authors



Boris Schapira

✉ @boostmarks
⌚ borisschapira
🌐 https://boris.schapira.dev
Developers, Reviewers, Translators



Colin Bendell

✉ @colinbendell
⌚ colinbendell
Reviewers



Brian Kardell

✉ @briankardell
⌚ bkardell
🌐 https://bkardell.com
Reviewers



Dave Crossland

✉ @davelab6
⌚ davelab6
🌐 https://fonts.google.com
Reviewers



Brian Rinaldi

✉ @remotesynth
⌚ remotesynth
🌐 https://remotesynthesis.com/
Analysts



Dave Smart

✉ @davewsmart
⌚ dwsmart
🌐 https://tamethebots.com/
Reviewers



Caleb Queern

✉ @httpsecheaders
⌚ queern
Reviewers



Dave Sottimano

✉ @dsottimano
⌚ dsottimano
🌐 https://opensourceseo.org/
Reviewers

	<p>David Fox Twitter: @theoboto GitHub: obto Website: https://www.lookzook.com Analysts, Editors, Project Leads, Reviewers </p>		<p>Estelle Weyl Twitter: @estellevw GitHub: estelle Website: http://standardista.com/ Reviewers </p>
	<p>Doug Sillars Twitter: @dougsillars GitHub: dougsillars Website: https://dougsillars.com Reviewers </p>		<p>Giovanni Punti Twitter: @giovannipunti GitHub: giopunti Reviewers </p>
	<p>Drewz GitHub: drewzboto Reviewers </p>		<p>Gokulakrishnan Kalaikovan GitHub: gokulkrishh Reviewers </p>
	<p>Durga Prasad Sadhanala Twitter: @dsadhanala GitHub: dsadhanala Developers </p>		<p>Greg Brimble Twitter: @GregBrimble GitHub: GregBrimble Website: https://gregbrimble.com/ Analysts </p>
	<p>Dustin Montgomery Twitter: @DustinMontSEO GitHub: en3r0 Website: https://dustinmontgomery.com Reviewers </p>		<p>Greg Wolf GitHub: gregorywolf Analysts </p>
	<p>Edmond W. W. Chan GitHub: edmondwwchan Website: https://edmondwwchan.github.io Reviewers </p>		<p>Henri Helvetica Twitter: @HenriHelvetica GitHub: henrihelvetica Authors </p>
	<p>Elika Etemad aka fantasai Twitter: @fantasai GitHub: fantasai Website: https://fantasai.inkedblade.net Reviewers </p>		<p>Ian Devlin Twitter: @iandevlin GitHub: iandevlin Website: https://iandevlin.com Authors </p>
	<p>Eric Bailey Twitter: @ericwbailey GitHub: ericwbailey Website: https://ericwbailey.design Reviewers </p>		<p>Jad Joubran Twitter: @JoubranJad GitHub: jadjojoubran Website: https://learnjavascript.online Reviewers </p>
	<p>Jamie Indigo Twitter: @Jammer_Volts GitHub: fellowhuman1101 Website: https://not-a-robot.com Authors </p>		



Jason Haralson

⌚ jrharalson
Analysts Reviewers



Laurent Devernay

🐦 @ldevernay
⌚ ldevernay
🌐 https://ldevernay.github.io/
Reviewers



Jason Pamental

🐦 @jpamental
⌚ jpamental
🌐 https://rwt.io
Authors



Lea Verou

🐦 @leaverou
⌚ LeaVerou
🌐 https://lea.verou.me/
Analysts Authors



Jens Oliver Meiert

🐦 @j9t
⌚ j9t
🌐 https://meiert.com/en/
Authors Reviewers



Leonardo Zizzamia

🐦 @Zizzamia
⌚ Zizzamia
🌐 https://twitter.com/zizzamia
Authors Reviewers



Jessica Nicolet

🐦 @jessica_nicolet
⌚ jessnicole
🌐 https://www.jessicanicolet.com/
Reviewers



Luca Versari

⌚ veluca93
Authors



Jonathan Wold

🐦 @sirjonathan
⌚ sirjonathan
🌐 https://jonathanwold.com
Reviewers



Lucas Pardue

🐦 @SimmerVigor
⌚ LPardue
🌐 https://lucaspardue.com
Reviewers



Julia Yang

⌚ jzyang
Reviewers



Lyubomir Angelov

🐦 @angelovcode
⌚ Super-Fly
Developers



Jyrki Alakuijala

⌚ jyrkialakuijala
Authors



Maedah Batool

🐦 @maedahbatool
⌚ MaedahBatool
🌐 https://maedahbatool.com/
Reviewers



Karolina Szczur

🐦 @fox
⌚ thefoxis
Authors



Mandy Michael

🐦 @Mandy_Kerr
⌚ mandymichael
🌐 https://mandymichael.com/
Reviewers



Katie Hempenius

🐦 @katiehempenius
⌚ khempenius
Analysts

**Manuel Matuzovic**

✉ @mmatuzo
⌚ matuzo
🌐 https://www.matuzo.at/
Reviewers

**Moritz Firsching**

⌚ mo271
🌐 https://mo271.github.io/
Authors

**Max Ostapenko**

✉ @themax_o
⌚ max-ostapenko
🌐 https://maxostapenko.com/
Analysts Developers

**Nate Dame**

✉ @seonate
⌚ natedame
Reviewers

**Michael DiBlasio**

⌚ mdiblasio
Authors

**Nicolas Goutay**

✉ @Phacks
⌚ phacks
🌐 https://phacks.dev/
Reviewers

**Michael King**

✉ @IPullRank
⌚ ipullrank
Authors

**Nicolas Hoizey**

✉ @nhoizey
⌚ nhoizey
🌐 https://nicolas-hoizey.com/
Reviewers

**Michelle O'Connor**

Designers

**Noah van der Veer**

✉ @noah_aaron_vdv
⌚ noah-vdv
Translators

**Miguel Carlos Martínez Díaz**

✉ @mcmd
⌚ mcmd
🌐 miguelcarlosmartinezdiaz
Translators

**Noam Rosenthal**

✉ @nomsternom
⌚ noamr
Reviewers

**Mike Bishop**

⌚ MikeBishop
Authors

**Nurullah Demir**

⌚ @nrllah
⌚ nrllh
🌐 https://internet-sicherheit.de/
Analysts Authors

**Minko Gechev**

✉ @mgechev
⌚ mgechev
🌐 https://blog.mgechev.com/
Reviewers

**Olu Niyi-Awosusi**

⌚ oluoluoxfree
🌐 https://www.opentagclosetag.com/
Authors

**Miriam Suzanne**

⌚ @MiriSuzanne
⌚ mirisuzanne
🌐 https://miriamsuzanne.com/
Reviewers

**Pascal Schilp**

⌚ thepassle
Reviewers



Patrick Meenan

✉ @patmeenan
⌚ pmeenan
🌐 https://www.webpagetest.org/
Reviewers



Renee Johnson

✉ @reneesoffice
⌚ ernee
🌐 https://reneesvirtualoffice.com/
Reviewers



Paul Calvano

✉ @paulcalvano
⌚ paulcalvano
🌐 https://paulcalvano.com/
Analysts, Developers, Project Leads,
Reviewers



Rick Viscomi

✉ @rick_viscomi
⌚ rviscomi
Analysts, Developers, Editors,
Project Leads, Reviewers



Pearl Latteier

✉ @pblatteier
⌚ pearlbea
Reviewers



Robin Marx

✉ @programmingart
⌚ rmarx
Authors



Pokidov N. Dmitry

✉ @otherpunk
⌚ dooman87
🌐 https://pixboost.com/
Analysts



Rocky Nebhwani

✉ @rnebhwanı
⌚ rockeynebhwanı
🔗 rockeynebhwanı
Authors



Praveen Pal

✉ @PraveenPal4232
⌚ PraveenPal4232
🌐 https://praveenpal4232.github.io/
Translators



Roel Nieskens

✉ @PixelAmbacht
⌚ RoelN
🌐 https://pixelambacht.nl/
Reviewers



Rachel Andrew

✉ @rachelandrew
⌚ rachelandrew
🌐 https://rachelandrew.co.uk/
Authors



Rory Hewitt

✉ @roryhewitt3
⌚ roryhewitt
🔗 roryhewitt
🌐 https://romche.com/
Authors



Raghu Ramakrishnan

✉ @raghuramakrishnan71
Analysts Authors



Sakae Kotaro

✉ @beltway7
⌚ ksakae1216
🌐 https://www.ksakae1216.com/archive/
Translators



Raph Levien

✉ @raphlinus
⌚ raphlinus
🌐 https://levien.com/
Authors



Sami Boukortt

⌚ sboukortt
Authors

**Saptak Sengupta**

✉ @Saptak013
⌚ saptaks
🌐 https://saptaks.website/
Developers

**Thomas Steiner**

⌚ tomayac
🌐 https://blog.tomayac.com/
Analysts Reviewers

**Sawood Alam**

✉ @ibnesayed
⌚ ibnesayed
🌐 https://www.cs.odu.edu/~salam/
Developers Reviewers

**Tim Kadlec**

✉ @tkadlec
⌚ tkadlec
Authors

**Shane Exterkamp**

✉ @Shane_Exterkamp
⌚ exterkamp
Editors Reviewers

**Tom Van Goethem**

✉ @tomvangoethem
⌚ tomvangoethem
Analysts Authors

**Shubhie Panicker**

✉ @shubhie
⌚ spanicker
Authors

**Tony McCreath**

✉ @TonyMcCreath
⌚ Tiggerito
🌐 https://websiteadvantage.com.au/
Analysts

**Simon Hearne**

✉ @simonhearne
⌚ simonhearne
🌐 https://simonhearne.com
Authors

**William Sandres**

✉ @hakacode
⌚ HakaCode
🌐 https://hakacode.github.io
Translators

**Simon Pieters**

✉ @zcorpan
⌚ zcorpan
Reviewers

**Yana Dimova**

⌚ ydimova
Analysts Authors

**Stefan Matei**

✉ @smatei
⌚ smatei
🌐 https://www.advancedwebranking.com/
Analysts

**Zuckjet**

✉ @Zuckjet
⌚ Zuckjet
Translators

**Sudheendra chari**

✉ @itsmesudheendra
⌚ sudheendrachari
🌐 sudheendrachari
Developers

**hemanth.hm**

✉ @gnumanth
⌚ hemanth
🌐 https://h3manth.com
Authors

**Tamas Piros**

✉ @tpiros
⌚ tpiros
🌐 https://www.fullstacktraining.com
Authors

**notwillk**

⌚ notwillk
Reviewers



rsheetter

 rsheetter

Reviewers