



# 2021

# 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 .....	63
Chapter 3: Markup .....	95
Chapter 4: Structured Data .....	129
Chapter 5: Media .....	157
Chapter 6: WebAssembly .....	195
Chapter 7: Third Parties .....	219

## Part II. User Experience

Chapter 8: SEO .....	249
Chapter 9: Accessibility .....	285
Chapter 10: Performance .....	331
Chapter 11: Privacy .....	359
Chapter 12: Security .....	393
Chapter 13: Mobile Web .....	431
Chapter 14: Capabilities .....	471
Chapter 15: PWA .....	493

## Part III. Content Publishing

Chapter 16: CMS .....	525
Chapter 17: Ecommerce .....	559
Chapter 18: Jamstack .....	603

## Part IV. Content Distribution

Chapter 19: Page Weight .....	631
-------------------------------	-----

Chapter 20: Resource Hints .....	643
Chapter 21: CDN .....	667
Chapter 22: Compression .....	687
Chapter 23: Caching .....	701
Chapter 24: HTTP .....	723

## Appendices

Methodology .....	753
Contributors.....	763

# Foreword

Three years ago I wondered to myself, *plenty of tools can tell me how well-built my website is, but where would I go to see the state of the web as a whole?* As sophisticated as the HTTP Archive dataset is, the answers it gives us can only be as useful as the questions we ask it. I'm a web developer, but I'm not an expert in all areas of web development—no one is expected to be! But collectively, we all have our own areas of expertise. Get enough of us together, and we can start to ask the right questions about the state of the web that the HTTP Archive can answer in really meaningful ways. That was the original idea behind the Web Almanac.

This year we're back with the third edition, which was made possible by the hard work of more than a hundred amazing people from the web community. I'd like to specifically call out a few people for whom this is their third consecutive year contributing: Barry Pollard, David Fox, Paul Calvano, Brian Kardell, Doug Sillars, Eric Portis, Thomas Steiner, Robin Marx, Alan Kent, and Abby Tsai. I owe every contributor an enormous debt of gratitude for volunteering their time to this project, but especially these 10 people who have been a part of it since the beginning.

The 2021 edition consists of a comprehensive lineup of 24 chapters, including two that we're excited to cover for the first time: Structured Data and WebAssembly. These new chapters help us expand the scope of the Web Almanac, which educates our reader base about a more diverse range of topics and equips even more specialized groups with actionable data. Ultimately, that's why we do it: we hope that our research can be utilized by the web community as a shared source of truth to meaningfully improve the ecosystem. If you find this resource as valuable as we do, we'd love it if you shared it with other people who are interested in the state of the web. Together, let's use this data as a forcing function for positive change.

— Rick Visconti, Web Almanac Editor-in-Chief



# Part I Chapter 1

# CSS



**Written by Eric A. Meyer and Shuvam Manna**

*Reviewed by Chris Lilley, Jens Oliver Meiert, Estelle Weyl, Brian Kardell, Adam Argyle, and Lea Verou*

*Analyzed by Rick Viscomi*

*Edited by Shaina Hantsis*

## Introduction

CSS (Cascading Style Sheets) is one of the three main pillars for building pages on the web—with HTML, used to define the structure; and JavaScript, used to specify behavior and interactions, completing the triumvirate.

Compared to last edition, the 2021 Web Almanac offers a deeper insight into how the use of CSS differs in the realm of what we all think we need versus what we actually see in production. As the calls for more robust CSS features and the challenge of centering a `<div>` with CSS kept making the rounds on our blog posts, conference talks, and Twitter chatter, pages around the web offered us vastly contradicting results, betraying the fact that CSS has, perhaps, become old enough to put more thought on staying stable instead of going wild with the zaniest of toys.

While CSS-in-JS adoption grew to 3% of all pages crawled (a 1 percentage point jump from last year), cutting-edge Houdini features are still mostly confined to tutorials and example galleries.

Responsiveness continued to be one of most engrossing priorities, with `max-width` and `min-width` being the top media queries, and `calc()` being the top CSS function most commonly in use to determine widths.

As users continue to throng to the web, let's jump into the data that would give us a better insight into how we have been faring in painting the internet—a place that is a second home, a workspace, a garage, or a rabbit hole for the rest of us.

## Usage



Figure 1.1. Distribution of stylesheet transfer sizes per page.

It isn't the heaviest component of most pages, but CSS—like the rest of the web—continues to grow in size from year to year. The median web page loads around 70 KB of CSS, and at the upper end, the average size is just over a quarter of a megabyte. Compared to 2020, the median total CSS weight rose about 7.9%, and the 90th percentile just under 7%, while preserving the pattern seen last year that mobile CSS is a little smaller than desktop CSS across all percentiles.

Not every page was so constrained: the page with the greatest CSS weight loaded 64,628 KB. The biggest mobile CSS weight seems positively svelte in comparison: only 17,823 KB.

As in 2020, it was found that page weight wasn't significantly driven by preprocessors. 17% of desktop pages and 16.5% of mobile pages included sourcemaps, up slightly from 15% last year. The consistent share of CSS including sourcemaps seems to indicate that the sourcemap share

is due more to build tool usage than sourcemap adoption, as we would expect to see much bigger year-over-year changes to sourcemap usage otherwise.

As for what kinds of sourcemaps were used, the numbers were largely consistent with last year:

Sourcemap type	2020	2021
CSS files	45%	45%
Sass	34%	37%
Less	21%	17%

Figure 1.2. Sourcemap types in 2021 versus 2020.

While this could be taken as evidence that Sass continues to gain ground over Less, the changes are small enough that it's difficult to call them significant, statistically or otherwise. Time, as always, will tell.

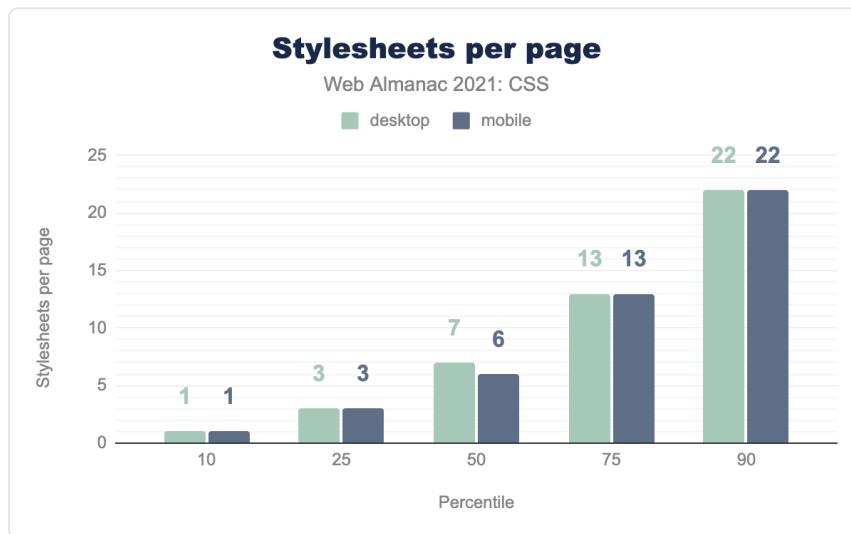


Figure 1.3. Distribution of the number of stylesheets per page.

In terms of the average number of stylesheets per page, whether embedded or external, the numbers this year are up only slightly from last year. The 50th through 90th percentiles went up by one each, while the 10th and 25th percentiles didn't budge.

# 2,368

*Figure 1.4. The largest number of external stylesheets loaded by a page.*

Incredibly, this year's record for the largest number of external stylesheets beat last year's by nearly a factor of two: 2,368 versus 1,379 in 2020. Whoever's done this, we beg you—combine some files and give your server a rest!



*Figure 1.5. Distribution of the total number of style rules per page.*

Number of stylesheets is one thing, but what about the number of actual style rules? Compared to last year, the lower percentiles rose a bit, while the highest barely budged. What is different in 2021 versus 2020 is that across nearly all percentiles, desktop pages have more rules on average than do mobile pages.

## Selectors and the cascade

Understanding cascade is an incredibly important part of working with CSS. Even more so for instances when you'd see that the styles you had written for an element are not working at all.

CSS offers a number of ways of applying styles to pages, from classes, ids and using the all-important cascade to avoid duplicating styles.

## Class names



Figure 1.6. The most popular class names.

Much like last year, the most popular class name on the web is `active`, and the `fa`, `fa-*` (the Font Awesome prefix), and `wp-*` (the WordPress prefix) class names make very strong showings. `selected` and `disabled` switched places in the lineup compared to last year, but the most heartening change was a 5% drop for `clearfix`, a sign that float-based layout continues to wane.

We were also heartened to see the placement of `sr-only-focusable`, which is a Bootstrap

accessibility feature. It causes an element to be placed off-screen, yet remains accessible to screen readers.

## IDs



Figure 1.7. The most popular ID names.

Pages continue to use IDs, and at about the same rate as seen in 2020. Even the list of popular ID names is consistent: `content` sits in the top spot at about 14% of pages, followed by `footer` and `header`. These latter two IDs dropped about a percent versus last year, which isn't really enough to say anything definitive about them other than, developers should replace them with the corresponding HTML elements `<header>` and `<footer>` whenever possible.

The IDs starting with `rc-` are part of Google's reCAPTCHA system, most versions of which are inaccessible in various ways<sup>1</sup>.

1. <https://www.w3.org/TR/turingtest/#the-google-recaptcha>

## Attribute selectors



Figure 1.8. The most popular attribute selectors.

The most popular attribute selector continues to be `type`, which is most likely to be used in selecting form controls like checkboxes, radio buttons, text inputs, and so on.

## Pseudo-classes and -elements

The ranking and distribution of both pseudo-classes and pseudo-elements was not greatly changed from the 2020 Web Almanac. A few rankings changed, but overall, things seemed highly static. Whether this represents a solidification of common practice, a snapshot of designer interests, or simply the nature of the analysis, is open to debate.



Figure 1.9. The most popular pseudo-classes.

Just as in 2020, the user-action pseudo-classes `:hover`, `:focus`, and `:active` took the top three spots, with all of them appearing in a minimum of two-thirds of all pages. Structural pseudo-classes put in a number of appearances, but one of the most interesting changes was `:not()`, the negation pseudo-class, becoming more popular than `:visited` and achieving a 50% share of pages.

One thing we did check specifically this year was the use of `:focus-visible`, a way to style elements in focus in a way that better matches user expectations. This capability landed in Chromium in 2020, Firefox in January 2021, and (as of publication) is available in Safari 15 behind an experimental flag. Likely reflecting its recent implementation status, it appeared in less than 1% of the pages analyzed. It will be interesting to see if that number changes over the next few years.

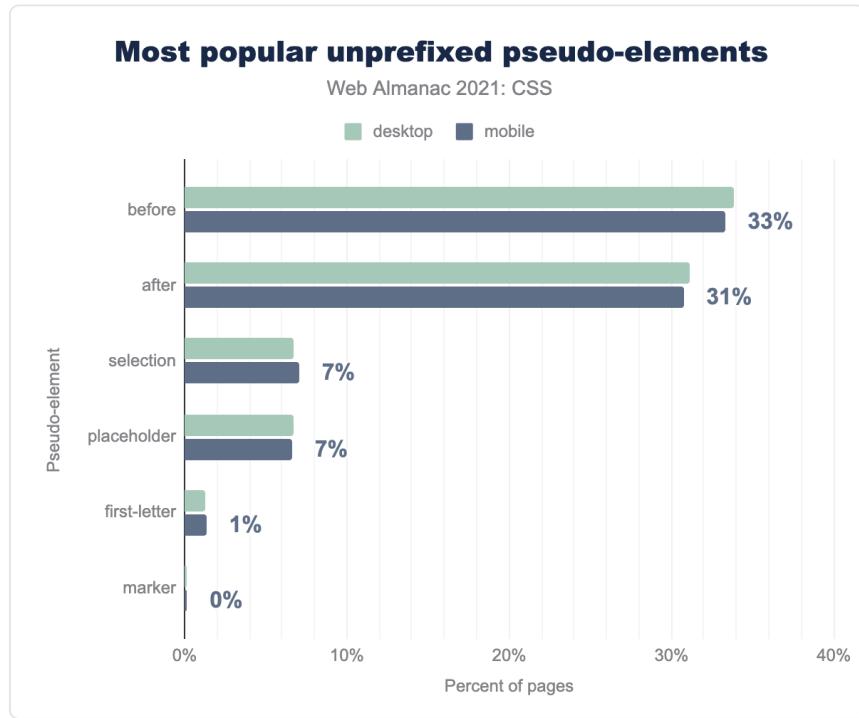


Figure 1.10. The most popular unprefixed pseudo-elements.

Most of the pseudo-elements in use are browser-specific ways of selecting things like specific interface components, parts of browser chrome, or highlighted text. Once we filtered those out, we found that `::first-letter` is used on a very small number of pages, but still many more than `::first-line`, which didn't make it onto the chart at all. `::marker`, a way of selecting list item markers like bullets or counters in an ordered list, has much less than 1% page share, yet still made it onto the list. We should note here that cross-browser support for `::marker` is relatively new<sup>2</sup> (October 2020). It will be interesting to see if use increases over the next few years.

2. <https://caniuse.com/css-marker-pseudo>

**!important**

Figure 1.11. Distribution of the percentage of page rules using `!important`.

That old battleaxe `!important` maintains a toehold on the web, with its share of marked rules hardly changing at all compared to the 2020 Web Almanac.

If that seems like a lot, hold on to your IDEs: we found a mobile page with 17,990 rules marked `!important`! That just edged out the most-important desktop page, which had 17,648 specificity-busting rules. We sincerely, truly hope these were the result of a script or preprocessor gone wrong.

As for what `!important` gets applied to, as with last year, it's `display`, with the rest of the chart falling in the same order as in 2020—with the exception of the last item on the chart, where `position` bumped off `float`.



Figure 1.12. The most popular properties targeted by `!important`.

## Selector specificity

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

Figure 1.13. Distribution of the median selector specificity per page.

Many CSS methodologies recommend that authors restrict themselves to single classes in order to squash all selectors' specificity into a single layer that is more easily managed. The BEM methodology<sup>3</sup>, for example, was found on 34% of all pages. The 10th percentile of median selector specificity shows further evidence of this type of thinking, where both desktop and

3. <https://en.bem.info/methodology/css/>

mobile specificity averages at (0,1,0). This is in line with last year's findings, as are nearly all the medians—with the exception of mobile's 25th percentile, which rose a little bit.

## Values and units

CSS provides multiple ways to specify values and units, either in set lengths or calculations based on global keywords.

### Lengths



Figure 1.14. The most popular length units.

Whatever you may think of pixel lengths, it's still the most popular length unit by far, appearing in about 71% of all pages. The second-place length unit, percentage, trailed pixels by an overwhelming distance.

<b>Property</b>	<b>px</b>	<b>&lt;number&gt;</b>	<b>em</b>	<b>%</b>	<b>rem</b>	<b>pt</b>
<code>font-size</code>	(▼1%) 69%	2%	(▼1%) 16%	(▼1%) 5%	(▲1%) 5%	2%
<code>line-height</code>	(▼5%) 49%	(▲3%) 34%	(▲1%) 14%	(▼1%) 2%	(▲1%) 1%	0%
<code>border-radius</code>	65%	(▼1%) 20%	3%	10%	(▲2%) 2%	0%
<code>border</code>	71%	(▲1%) 28%	2%	0%	0%	0%
<code>text-indent</code>	(▼1%) 31%	(▲1%) 52%	8%	(▼1%) 8%	0%	0%
<code>gap</code>	(▼8%) 13%	(▲2%) 18%	(▼1%) 0%	0%	(▲7%) 69%	0%
<code>vertical-align</code>	(▼11%) 18%	12%	(▲11%) 66%	4%	0%	0%
<code>grid-gap</code>	(▲3%) 66%	(▼1%) 10%	9%	(▼1%) 0%	(▼2%) 14%	0%
<code>padding-inline-start</code>	(▼7%) 26%	(▲2%) 7%	(▲4%) 66%	0%	0%	0%
<code>mask-position</code>	0%	0%	(▼1%) 49%	(▲1%) 51%	0%	0%
<code>margin-inline-start</code>	(▼7%) 31%	(▲5%) 51%	(▲1%) 15%	(▲2%) 2%	1%	0%
<code>margin-block-end</code>	(▲1%) 5%	(▲7%) 38%	(▼9%) 56%	0%	(▲1%) 1%	0%

Figure 1.15. Distribution of length types per property.

Where things become interesting is in the breakdown of exactly how the various length units are used. To pick one example, the most common length unit used on `line-height` is pixels, followed by `<number>` values (which includes all instances of unitless zero length values).

`em`s are the most popular length unit for `vertical-align` and `padding-inline-start`.

The positive and negative figures given in parentheses next to the figures in this table show change from 2020 results. In almost every property we analyzed, pixels became less popular as compared to the uses of other length units, with just two exceptions. The biggest change by far was in `vertical-align`, with an 11-point shift from pixels to `em`s as the unit of choice when the supplied value was a length, as opposed to a keyword like `baseline`.



Figure 1.16. The most popular font-relative length units.

Although `em` maintains a huge dominance over `rem` when it comes to sizing fonts, there are signs of change: there was a seven-point swing from `em` to `rem` between 2020 and 2021.



Figure 1.17. The units (or lack thereof) used on zero-length values.

There are a few properties that allow bare `<number>` units (e.g., `line-height`), but

`<length>` values have a special case where a length of zero does not require a unit. When we looked at all zero-length values, almost 88% of them omitted the unit. Nearly all of those zero lengths that included a unit used pixels (`0px`). This was a nice result to see, since any length of zero doesn't need a unit and including one is fairly pointless. We hope the share of unitless zero values will grow in the future.

## Calculations

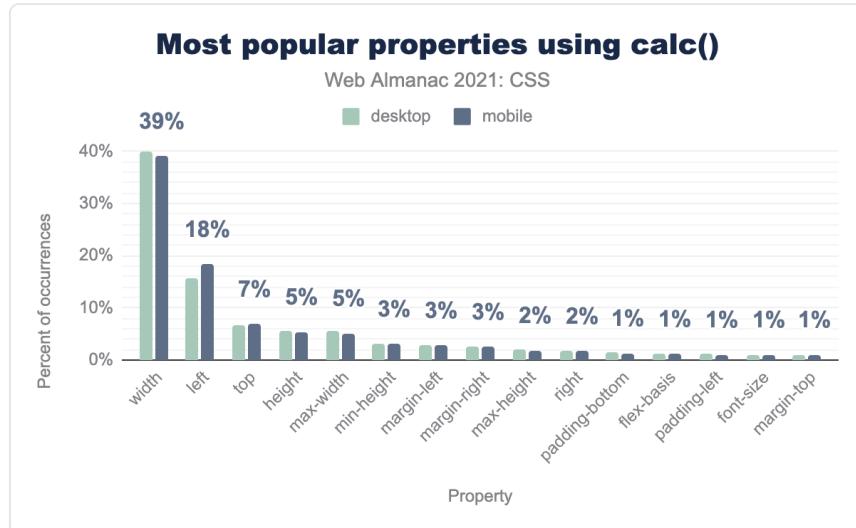


Figure 1.18. The most popular properties using `calc()` functions.

As in past years, the most popular usage of `calc()` is to set widths, although the share of `calc()` values in `width` dropped a full 20 points as compared to 2020. This seems most likely to reflect an expansion of `calc()` use in other properties, rather than a contraction of its use for `width`.

## Most popular units used in calc()

Web Almanac 2021: CSS

desktop mobile

Figure 1.19. The most popular length units used in `calc()` functions.

Although pixel units didn't shift at all in terms of their usage in calculations, percentages lost a bit of ground compared to the long tail of other units, falling four points since 2020.

## Most popular operators used in calc()

Web Almanac 2021: CSS

desktop mobile

Figure 1.20. The most popular operators used in `calc()` functions.

As with last year, when it comes to calculation operators, subtraction is the clear favorite, and

barely shifted its share of usage. There were much bigger changes in the second and third spots, where addition vaulted ahead of division, gaining six points while division dropped a similar amount.



Figure 1.21. The number of unique units used in `calc()` functions.

`calc()` values remain relatively simple, with the overwhelming preponderance of calculations using two different units, such as to subtract pixels from the calculated result of a percent value. A total of 99% of all `calc()` expressions use either one or two unit types.

## Global keywords



Figure 1.22. Usage of global keyword values.

The use of global keywords such as `initial` rose significantly as compared to the 2020 Web Almanac. While `inherit` only gained a couple of points, `initial` rose about eight points, and `unset` around 10 points. Even `revert` managed to lift itself up a point.

## Colors



Figure 1.23. The most popular color value formats.

Despite the availability of a wide number of color value types, the `#RRGGBB` syntax that has been with us since the days of Netscape 1.1 is still used in half of all color declarations. The CSS innovation of the `#RGB` shorthand came in second, at a quarter of color values. In other words, a solid 75% of all color values are expressed using hexadecimal RGB syntax. The third-place format, `rgba()`, points to the likely reason authors go beyond the classic hexadecimal format: to get access to alpha values. (Indeed, though both their shares are tiny, `hsla()` is more popular than `hsl()`, just as `rgba()` is much more common than plain `rgb()`.)

In color formats where the value has historically used commas inside a functional syntax—for example, `rgba(0, 0, 0, 1)`—authors may now drop the commas and separate colors from alpha with a slash (thus, `rgb(0 0 0 / 1)`). Since 2020, this comma-less syntax has doubled its usage share, going from 0.12% to 0.25% of all functional color syntax.

	<b>Keyword</b>	<b>Desktop</b>	<b>Mobile</b>
□	<i>transparent</i>	82.24%	82.93%
□	<i>white</i>	7.97%	7.59%
■	<i>black</i>	2.44%	2.29%
■	<i>red</i>	2.23%	2.17%
■	<i>currentColor</i>	1.94%	2.03%
■	<i>gray</i>	0.68%	0.64%
■	<i>silver</i>	0.56%	0.55%
■	<i>grey</i>	0.39%	0.37%
■	<i>green</i>	0.32%	0.31%
■	<i>blue</i>	0.15%	0.12%
□	<i>whitesmoke</i>	0.12%	0.11%
■	<i>orange</i>	0.12%	0.10%
■	<i>lightgray</i>	0.08%	0.08%
■	<i>lightgrey</i>	0.07%	0.07%
■	<i>yellow</i>	0.07%	0.06%
■	<i>gold</i>	0.04%	0.03%
■	<i>magenta</i>	0.03%	0.03%
■	<i>Background</i>	0.02%	0.03%
■	<i>Highlight</i>	0.02%	0.03%
■	<i>pink</i>	0.03%	0.03%

Figure 1.24. The most popular named-color keyword values.

In the realm of just the named colors, `transparent` is still the faraway favorite, with around 82% of all named color keyword usage. The familiar and comfortable `white`, `black`, and `red` total another 12% or so, and `currentColor` comes in fifth with a half-percent rise over its 2020 numbers.

In last year's Web Almanac, there was a note about "the once-deprecated—now partially undeleted—system colors like `Canvas` and `ThreeDDarkShadow`" being just barely in use. This is still true, but oddly, there are now two such values in the top 20 instead of just one (`Highlight`). That said, both occur in the realm of tiny, tiny numbers of pages, so such shifts are probably unremarkable.



*Figure 1.25. Percentage of display-p3 colors that lie outside the sRGB space.*

The usage of the display-p3 color space remains about as vanishingly small as was found in 2020, probably because it's only supported in Safari (both desktop and mobile) as of this writing. Desktop and mobile use roughly tripled, to 90 and 105 pages, respectively. In the cases where `color(display-p3)` was used, it was with good reason: 79% of the colors expressed using display-p3 on mobile were colors that cannot be represented in the sRGB color space. Until the `color()` function becomes more widely supported by browsers, the web will remain stuck in sRGB, which permits about two-thirds of the colors that screens can actually display.

## Images

They say a picture is worth a thousand words, but byte wise, they often cost an order of magnitude or two more. While there are a myriad of approaches to embedding images with JavaScript, or include them with the HTML scaffolding, here we looked at how CSS-loaded images are used.

### Formats of images in CSS

First, here's a breakdown of the image formats we looked for, and how often each format appeared:



Figure 1.26. Distribution of the formats of external images loaded via CSS.

PNG was the clear favorite, with a surprisingly close clustering of GIF, SVG, and JPG following behind. The fairly new WEBP format accounted for only 3.7% of images loaded by CSS, and the tiny slice at the top corresponds to unrecognized values and the ICO format.

We did not attempt to determine whether any of the images were animated.

Please also note that this analysis only covers the images loaded by CSS: we did not check the HTML to see what was being loaded there. Thus, the following results cannot be taken as a metric of how heavy web pages are, or even how heavy CSS is or is not. It can only show how much CSS-loaded images contribute to a page's total weight.

## Number of images in CSS

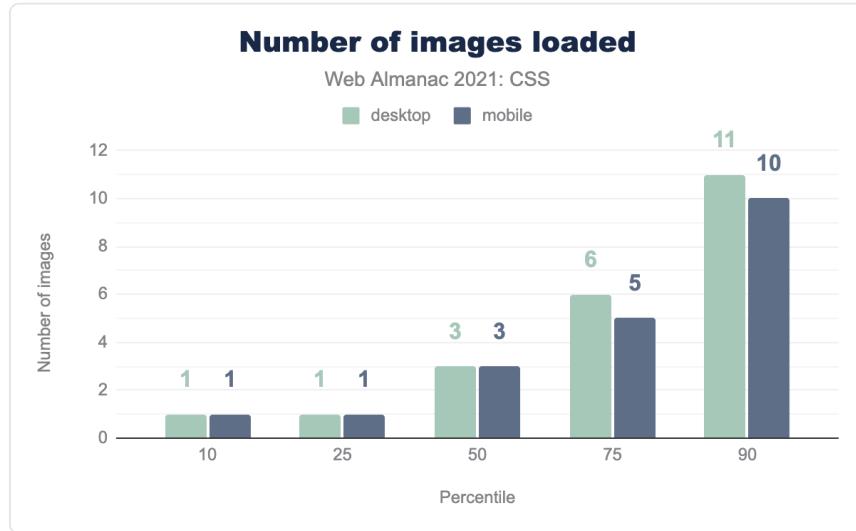


Figure 1.27. Distribution of the number of external images loaded via CSS.

We found that most CSS doesn't result in a lot of image loads: the lower two percentiles came in at one image each, and even the 90th percentile hovered around 10 images, across all image types.

**6,089**

Figure 1.28. The largest number of external images loaded by a page's CSS.

We did find one site where the desktop CSS loaded 6,088 PNG images. The mobile version of the site actually added an image, bringing it to 6,089 PNGs. We hope they were all small and color-indexed for efficiency's sake.

## Weight of images in CSS

The number of images is one thing, but how much they weigh is at least as important—loading a single 10 MB background is worse than loading ten 100 KB pictures, after all, even with server compression factored in.



Figure 1.29. Distribution of the total weight in KB of external images loaded via CSS.

All told, things were not as bad as we'd feared going in: the median page's CSS loads a total of 16 KB or so in images. It was also encouraging to see that overall, mobile image loading via CSS was consistently a bit lower than desktop—a sign that CSS developers do keep the limitations of mobile contexts at least somewhat in mind.

# 314,386

Figure 1.30. The heaviest total weight of images loaded via CSS, in KB.

Sometimes, anyway. We did find a page where the total weight of the images loaded by CSS was a gargantuan 314,386.1 KB—a third of a gigabyte.

<b>Percentile</b>	<b>JPG</b>	<b>PNG</b>	<b>GIF</b>	<b>(other)</b>	<b>SVG</b>	<b>WebP</b>
10	4.5	0.7	0.5	0.3	0.4	1.7
25	28.2	2.2	1.7	0.3	0.6	14.2
50	114.3	7.0	3.7	0.3	1.7	39.6
75	350.7	36.4	8.3	48.1	5.4	133.9
90	889.3	173.6	13.0	229.2	20.0	361.8

Figure 1.31. Distribution of the total weight in KB of external images loaded via CSS on mobile pages, by image format.

When we broke down the image weights by format, we discovered a fascinating tidbit: at the 90th percentile, GIF images were actually lighter on average than even SVG files.

It was also interesting, though perhaps not surprising, that the heaviest image format was JPG. This is likely because JPG is favored for those big splashy photographs one so often sees across the tops of home pages and so forth, and even with compression and other optimization tricks, all those pixels do add up.

## Gradients

<b>Property</b>	<b>Desktop</b>	<b>Mobile</b>
<code>background</code>	62%	62%
<code>background-image</code>	62%	61%
<code>-webkit-mask-image</code>	5%	5%
<code>--*</code>	1%	1%
<code>mask-image</code>	1%	1%
<code>border-image</code>	1%	1%

Figure 1.32. Percentage of properties given gradient image values.

The share of pages using CSS gradients was roughly the same as last year: 77% of desktop pages and 76% of mobile pages. The properties on which they were used did change, however: while still the overwhelming favorites, `background` and `background-image` were the

properties to which about 62% of gradients were assigned.



*Figure 1.33. The most popular types of gradient image values.*

Linear gradients continue to be the clear favorite, maintaining the 5-to-1 lead over radial gradients seen in the 2020 Web Almanac<sup>4</sup>.

When prefixed versions of gradients (e.g., `-webkit-linear-gradient`) were included, the resulting graph looked basically the same as last year's.

Some other things we found in analyzing gradient values:

- The median number of color stops in gradients is just two, except at the 90th percentile, where the four stops was the median.
- Hard color stops—that is, gradients where two color stops were placed at the same position—occurred in just over half of all gradients.
- Color-stop interpolation (a.k.a. “midpoints”) were used in 21% of all gradient instances.

4. <https://almanac.httparchive.org/en/2020/css>



Figure 1.34. The linear gradient with the most color stops.

We also saw a dramatic reduction at the top end of gradient complexity. Last year, the gradient with the largest number of color stops had 646 stops. This year, the winner had only 81 color stops.

## Layout

We have come a long, long way from using tables to create layouts on the web to a time when we have a number of options to choose from—Flexbox, Grid, and Multicolumn, as well as old chestnuts like floats, positioning and even CSS table properties. We did a simple search of stylesheets to see which property and value combinations were present, and came up with the following figures.



Figure 1.35. The most commonly-declared layout types.

Note that this doesn't chart primary layout methods—we are not claiming here that 93% of the pages we analyzed are laid out using absolute positioning! Rather, what the chart says is that `position: absolute` appeared in the styles for 93% of the page we analyzed, even if that was just to put an icon in a corner or place bits of content `-9999px` offscreen. Similarly, `display: grid` may have appeared in 36% of page's styles, but that doesn't mean 37% of all pages are Grid pages, just that the combination appeared somewhere in the stylesheet.

The rest of this section is where more in-depth analyses were done, looking not just for property-value combinations, but for evidence of actual usage on pages.

## Flexbox and Grid adoption



Figure 1.36. Adoption of Flexbox and Grid layout on mobile devices.

The adoption of Flexbox and grid continues to grow. In 2019, Flexbox adoption was 41%; in 2020, it was 63%. This year, Flexbox hit 71% on mobile and 73% on desktop. Grid, in the meantime, has been doubling each year of the Web Almanac, from 2% to 4% and now 8%. Note that, in contrast to the previous section, what is measured here is the percentage of pages that are actually using Flexbox or Grid for layout, as opposed to the pages that simply have some sort of Flexbox or Grid property in their stylesheet.

## Usage of different Grid layout techniques

Digging into the various Grid properties, we discovered a few interesting patterns.

- About 15% of all Grid pages used `grid-template-areas` to define named areas of the grid.
- When we looked for square brackets in Grid templates, which would indicate the presence of named Grid lines, we found a little fewer than 10,000 pages out of the seven million or so analyzed.

We also analyzed Flexbox layouts to see which ones set the flex grow and shrink values to zero, and then set all the flex item widths to be something static, like percentage or pixel widths. These are referred to as “Grid-like Flexbox,” and we found that just over a quarter of all Flexbox layouts met these criteria. Given the complexity of the analysis, it is entirely possible that we missed many cases. Still, it seems clear that designers are strongly interested in grid-style layouts, and this could drive migration to Grid in the coming years.

## Multicolumn

A large, bold, dark blue percentage sign, representing 20%.

*Figure 1.37. The percentage of pages using multicolumn layout.*

Even though multicolumn layout is a bit fraught on the web, where it can force users to scroll down to the bottom of a column and then back up to the top of the next column, we detected multicolumn use on 20% of the pages we analyzed, which is a 5% rise over the 2020 Web Almanac. We continue to be surprised to see it on so many pages, and even more surprised to see its adoption increasing.

## Box sizing



Figure 1.38. Distribution of the median number of `border-box` declarations per page.

The principles of the original W3C box model continue to be rejected: when we looked to see how many pages were using `box-sizing: border-box`, it was an overwhelming 90%, up around 5% from 2020. Almost half of all pages analyzed apply border-box sizing to every element on the page via the universal selector (`*`). This “one sizing fitted to all” approach may help explain why the median number of `border-box` declarations per page is so low across the bottom three percentiles.

In addition, about a quarter of pages apply `box-sizing` to checkboxes and radio buttons.

## Transitions and animations

Animations continue to be widely used, with the `animation` property appearing on 77% of all mobile and 73% of all desktop pages analyzed. It’s even more popular cousin, `transition`, is used on 85% of all mobile and 90% of all desktop pages.

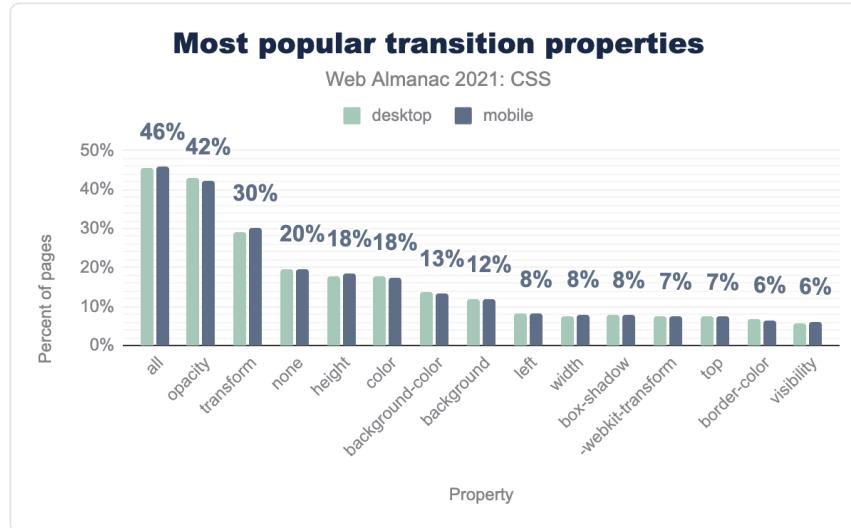


Figure 1.39. The most popular properties given transition effects.

Among those transitions, the most common application is to all animatable properties<sup>5</sup> using the `all` keyword (whether explicitly or by default), which occurred in 46% of the analyzed pages. Just behind that is `opacity`, at 42% of all pages containing transitions.

5. [https://developer.mozilla.org/docs/Web/CSS/CSS\\_animated\\_properties](https://developer.mozilla.org/docs/Web/CSS/CSS_animated_properties)



Figure 1.40. Distribution of transition durations.

We took a look at the duration and delay times of those transitions. Even at the 90th percentile, the median transition duration was just half a second.



Figure 1.41. Distribution of transition delays.

The highest median transition delay was 1.7 seconds, but even more interestingly, the 10th

percentile median delay was about not quite one-third of a negative second, indicating that a large number of transitions are started partway through the resulting animation (which is what negative delays cause to happen).

A closer look at the range of transition durations and delays revealed some seriously lengthy spans of time. The largest duration value we found was 9,999,999,999,999,996 seconds, which corresponds to almost 317 million years. Put another way, if that duration were used in a horizontal scroll transition of *If the Moon Were Only 1 Pixel*<sup>6</sup>, it would take just over two centuries to scroll to the right by a single pixel. This, however, pales in comparison to the longest transition delay we found: a value in milliseconds that equals not quite 31.7 *quintillion* years.



Figure 1.42. Adoption of transition timing functions.

As for the timing functions used during the transitions, the clear leader is the default value, `ease`. There's a virtual tie for second between `ease-in-out` and `linear`, but the surprise was our fourth-place finisher, `cubic-bezier`. This seems most likely to come from a library or some sort of tool, because while it's possible to learn how to construct cubic Bézier curves by hand, very few people bother to do so (nor is there much reason why they should).

Okay, but what kinds of animations are being performed? To determine this, we classified various animation labels by the type of animation being performed. For example, animations labeled `fa-spin`, `spin`, `spinner-spin`, and so on were classified as "rotate" animations, and these were the most popular.

6. [https://www.joshwirth.com/dev/pixelspace/pixelspace\\_solarsystem.html](https://www.joshwirth.com/dev/pixelspace/pixelspace_solarsystem.html)

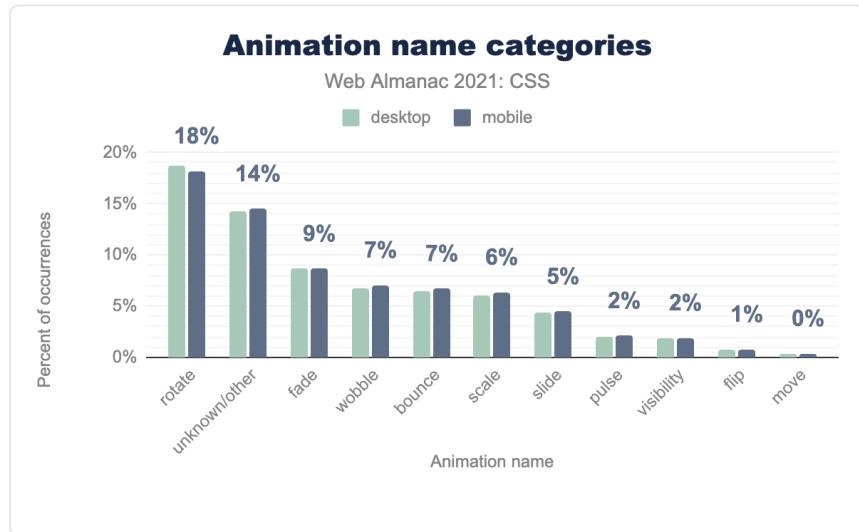


Figure 1.43. The most popular types of animation.

One reason for the high ranking of “unknown/other” is the animation label `a`, which was around 6-7% of all named animations. (The most likely companion to these, `b`, had a 2% prevalence.)

The weak showing of “move” and “slide” style animations might seem surprising but remember: these are specifically types of `animation`. Transitions driven by the `transition` property are not represented in this sample. It is highly likely that many simple movements (and fades) are handled with transitions, and `animation` is reserved mostly for more complex effects.

## Responsive design

Making a site that copes well with all the different screen sizes wherein you can now browse the web has become significantly easier with the advent of built-in tools like Flexbox and Grid, which are further enhanced by using media-queries.

### Media features in use

When authors build their media queries, they most often test the width of the viewport. `max-width` and `min-width` were the most popular queries by far, the same as in 2020. There was no ranking change in the third and fourth place results either.

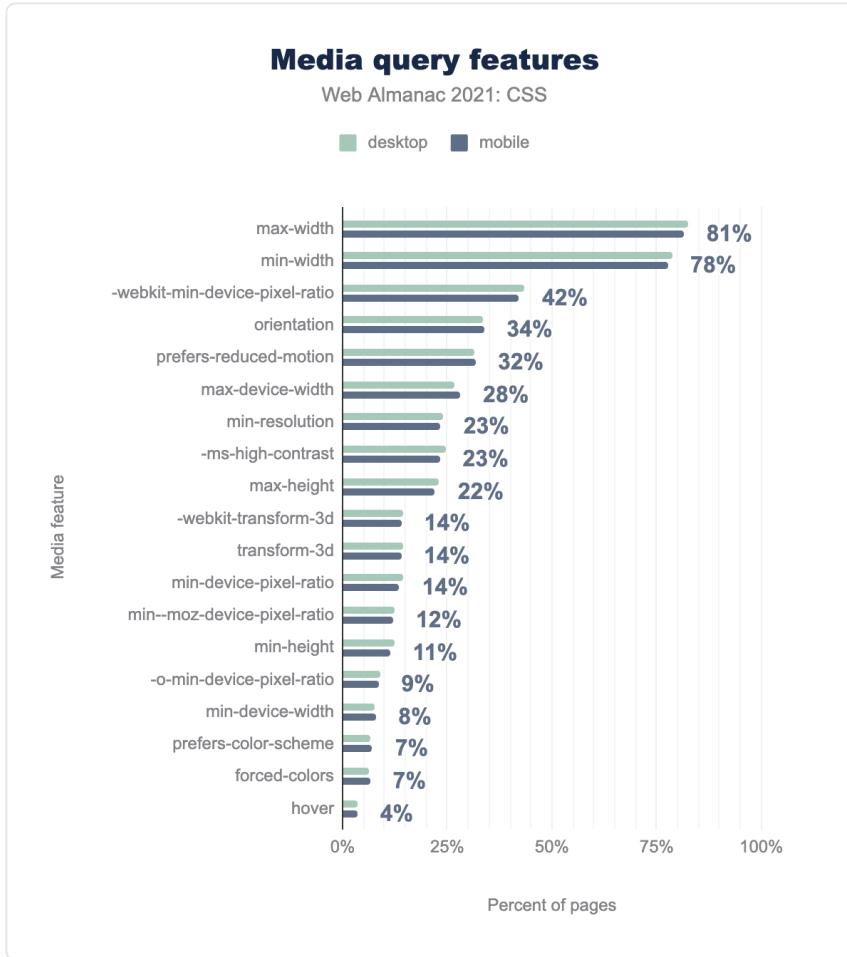


Figure 1.44. The most popular features used as media queries.

Where we did see a notable change was in the ranking of the `prefers-reduced-motion` query. This query placed 7th in 2020, with a share of 24%; this year, with a share of 32%, it's up to fifth, where it just missed edging out `orientation`.

We also saw newcomers come and go at the bottom of the list. `pointer`, a query which checks to see if the display device's primary input mechanism is a pointing device such as a mouse and which placed 19th last year, fell off the chart as it slipped to 21st place. The `hover` media feature, on the other hand, entered the chart at 20th place. `hover` is used to test if the display device's primary input mechanism can cause a hover state in elements on the page.

Both queries have a similar aim, which is (put simply) to figure out if the device being used to display the page is mouse-driven or not. Combined with a mobile-first design philosophy, where desktop styles are added to override the default mobile styling, one can see how queries like `pointer` or `hover` would be useful. While it's too soon to say if one or the other will become dominant, the trends this year swung toward `hover`.

This year also saw the debut of `prefers-color-scheme`, coming in at 7%. This may be due to iOS devices adding dark mode support since last year's report, but in any event, it's good to see that designers are starting to take color scheme preferences into account.

## Common breakpoints

As in 2020, the most common breakpoints by far are at 767 and 768 pixels, which correspond suspiciously well with the resolution of an iPad in portrait mode. We found `767px` was overwhelmingly used as a maximum-width breakpoint and only rarely as a minimum-width value. `786px`, by contrast, was quite often used as both a minimum and maximum breakpoint.



Figure 1.45. The most popular media query breakpoints.

Beyond the 767-768 range, the next most popular breakpoints were at 600 and 1,200 pixels, and close behind that was 480 pixels.

Lest you think we converted all the breakpoint queries to pixels, we're sorry to say we did not: these are the straight values from stylesheets. Out of all the breakpoints we analyzed, the first non-pixel value on the list is `48em`, which came in at 76th on the ranking list, appearing in 1% of

desktop and 2% of mobile styles. The next em-based value, `40em`, is found in 85th place.

## Properties inside media queries

So, what do authors actually style inside these media query blocks? The most often property to set is `display`, followed closely by `color`, `width`, and `height`.



*Figure 1.46. The most popular properties to be changed via media queries.*

One of the most notable changes between 2020 and 2021 was the fall of `font-size` as a property set inside media blocks. In 2020, it appeared in 73% of all media blocks, placing fifth on the list. This year, it appeared in around 60% of all media blocks, coming in 12th on the list.

`margin-right` and `margin-top` had even bigger falls, going from 8th and 9th to 25th and 17th, respectively. These sorts of shifts strongly imply a change in a common framework or piece of software—a change in the default WordPress theme would be one example, though we cannot say if this is the exact source of the change.

## Feature queries

Feature queries (`@supports`) continue to grow in usage. In 2019, 30% of pages were found to use them, and last year it was 39%. In 2021, almost 48% of pages are using feature queries to decide which CSS to apply in what contexts.

So, what do authors condition CSS upon? Sticky positioning was far and away the most popular query, accounting for over half of all feature queries.



*Figure 1.47. The most popular CSS features to be queried with `@supports`.*

Only 3% of feature queries checked for Grid support, which translates to 261,406 pages querying Grid support. Given that we found grid layout in use on 2.7 million mobile pages and 2.3 million desktop pages, if our numbers are accurate, it appears that the vast majority of Grid layouts are deployed without fallbacks.

## Custom properties



Figure 1.48. Change in custom property usage, 2019-2021.

Over the three years of the Web Almanac, custom properties (also known as CSS variables) have seen one of the greatest surges in usage. In 2019, usage was around 5% of all sites, and last year that had shot up to nearly 20% mobile and 15% desktop. This year, we found custom properties being defined on 28.6% of all mobile pages, and 28.3% of desktop pages. Even more, we found that 35.2% of mobile and 35.6% of desktop pages contained at least one `var()` function value.

## Naming

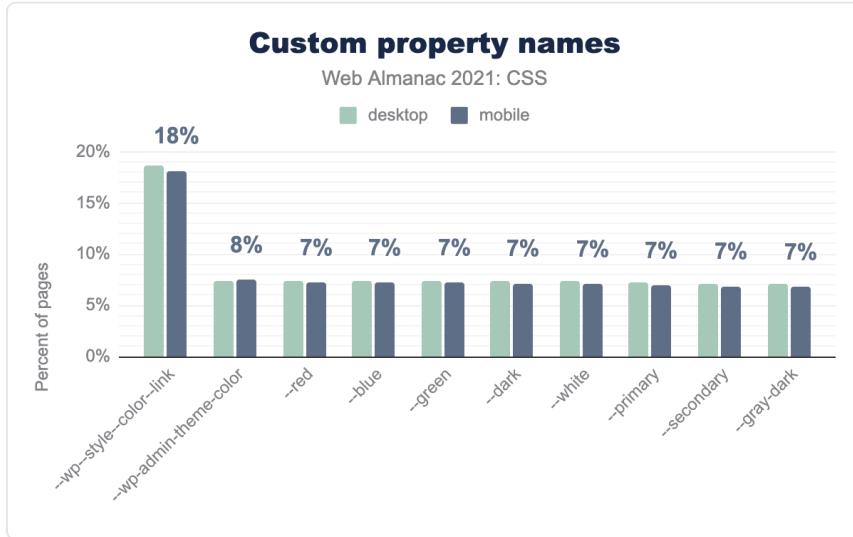


Figure 1.49. The most popular custom property names.

The first thing we checked was, “What are developers calling their custom properties?” As it turned out, the prevalence of WordPress came out here, with the top entry being a link-coloring custom property defined by the WP core.

After that, a lot of color names were found. It might seem odd that anyone would need to define a custom value for `--blue` when the named color `blue` is sitting right there, but in practice, developers are assigning custom shades to their basic color names. So rather than `--blue: blue`, we see declarations like `--blue: #3030EA`.

## Usage



Figure 1.50. The most popular properties to be given a custom-property value.

In addition to all the custom properties named after colors, the four most popular properties to be the recipients of custom-property values (using the `var()` function) are all setting color in one way or another.



Figure 1.51. Distribution of types of custom property values.

Each custom property gets a CSS value of one type or another. For example, `--red: #EF2143` is assigning a color value to `--red`, whereas `--multiplier: 2.5` is assigning a number value. We found that the most popular value type was colors, followed by dimensions (lengths), and then fonts families, whether singly or in groups.

## Complexity

It's possible to include custom properties in the values of other custom properties. Consider this example from the 2020 Web Almanac:

```
: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 */
}
```

As the comments in the previous example show, the more of these sub-references are chained together, the greater the *depth* of the custom property.



Figure 1.52. Distribution of median custom property depth.

Perhaps unsurprisingly, the clear majority of custom properties had a value depth of zero: they did not include the values of other custom properties in their own values. Nearly a third have one level of depth, and beyond that, there are almost no custom-property values with a depth of two or more.

As in 2020, we also checked the selectors in which custom-property values were used. Almost 60% were set on the root element (using either the `:root` or `html` selectors), and around 5% were applied to the `<body>` element. The rest were applied to some descendant of the root element other than `<body>`. This means around two-thirds of all custom properties are used as what are, in effect, global constants. This is in line with the results seen last year.

## Internationalization

English is written horizontally, and the characters are read from left to right. But for languages such as Arabic, Hebrew and Urdu, among others, are written right to left and then there are languages and scripts—such as Mongolian, Chinese, and Japanese—which can be written in vertical lines, from top to bottom. Owing to this, things can get quite complicated. Both HTML and CSS provide ways to handle this.

## Direction

Text direction can be explicitly enforced using the CSS property `direction`. We found it in use on the `<html>` element in 11% of all pages, and on the `<body>` element on 3% of pages. (Note that there may be overlap there, as we did not check for duplicate results.)

Of those pages that used CSS to set direction, 92% of `<html>` elements and 82% of `<body>` elements were set to `ltr` (left-to-right). Overall, we found `rtl` (right-to-left) used on only 9% of pages that set a direction in CSS. This is more or less to be expected, given that most languages are not right-to-left.

## Logical and physical properties

Another CSS feature useful for internationalization are the “logical” properties like `margin-block-start`, `padding-inline-end`, and so on, as well as values such as `start` and `end` for properties like `text-align`. These properties and values allow box features to be tied to the direction of text flow, rather than physical directions like top, right, bottom, and so on.

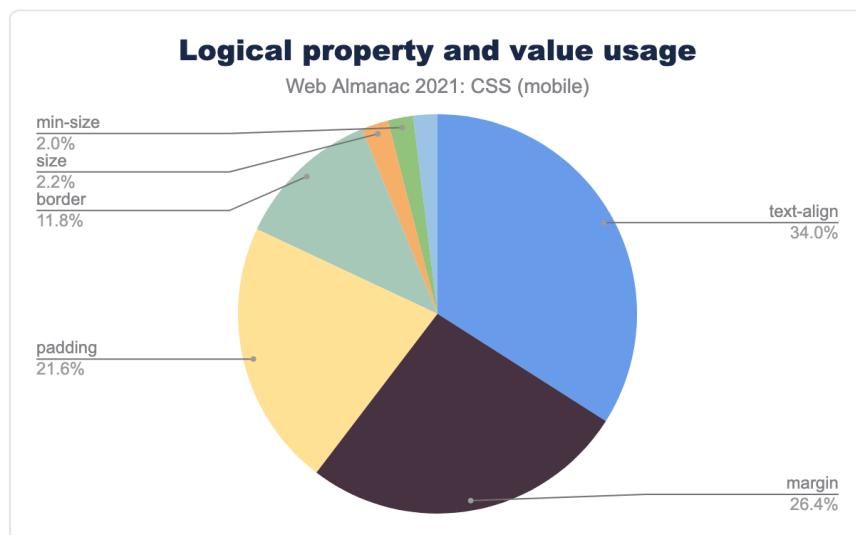


Figure 1.53. Distribution of property types of logical properties.

As of mid-2021, only 4% of pages were found to be using logical properties of any kind. Of the pages that did, about 33% were using it to set `text-align` to `start` or `end`. Another 46% or so (combined) were setting logical margins and padding. Again, note that there could be overlap in these figures.

## Ruby

In addition to directionality and logical features, CSS also offers internationalization support via CSS Ruby, a collection of properties used to affect the layout of interlinear annotation, which are short runs of text alongside the base text. Its usage is vanishingly small: only 8,157 desktop pages and 9,119 mobile pages were found to be using it—less than 0.1% of all pages analyzed.

## CSS and JS



*Figure 1.54. Distribution of CSS-in-JS libraries.*

While the topic of “CSS in JS” is good for at least a Twitter flame war or two, its use in the wild continues to be very small. This year, we found that about 3% of pages are using some form of CSS-in-JS, up from 2% in 2020. Furthermore, nearly all of it comes from libraries built for the purpose, and more than half of that usage is from the Styled Components library.

## Houdini

In some ways, CSS Houdini represents the opposite of the CSS-in-JS approach: it allows authors to mix a little JS into their CSS. Perhaps in part due to slow implementation<sup>7</sup> (in browsers that

7. <https://ishoudinireadyyet.com/>

aren't based on Blink) of core parts of the specification, Houdini has struggled to find its feet. We find that it's effectively not used on the open web in 2021: only 1,030 desktop pages and 1,175 mobile pages show evidence of animated custom properties, a feature of Houdini. This is a threefold increase over the 2020 findings, but it looks like it will still be some time before Houdini finds an audience.

## Meta

In this section, we take a look at more generic concepts in CSS, such as how often declarations are repeated or what kinds of mistakes authors make in writing their CSS.

### Declaration repetition

In the 2020 Web Almanac, analysis was done to determine the amount of "declaration repetition"—a metric meant to roughly estimate the efficiency of a stylesheet by determining how many declarations used the same property and value, and how many were unique within the page's styles.

The 2021 figures are in and appear to show a slight drop in the median amount of repetition across all percentiles.

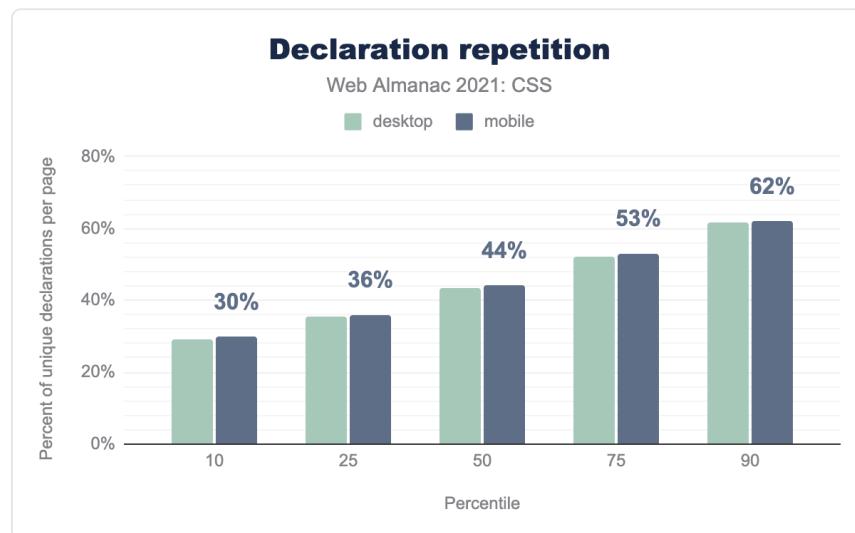


Figure 1.55. Distribution of repetition of declarations per page.

The degree of this drop is on the order of 2% for the 10th, 50th, and 90th percentiles, so it is entirely possible this is statistical noise. The only way to tell would be to continue the analysis in future years and chart the long-term trends.

## Shorthands and longhands

There are many parts of CSS where a collection of very specific properties are also covered by a single “umbrella” property that can set the more specific properties’ values in a single declaration. `font`, for example, encompasses the values of `font-family`, `font-size`, `line-height`, `font-weight`, `font-style`, and `font-variant`. The umbrella property `font` is what’s called a “shorthand” property, because it allows authors to set a number of things in a kind of shorthand. The corresponding specific properties (e.g., `font-family`) are referred to as “longhand” properties.

### Shorthands before longhands

If an author mixes shorthand properties like `background` and longhand properties like `background-size` in a stylesheet, it is always best to have the longhands come after the shorthands. We looked at instances where authors did this to see which longhands were most common.



Figure 1.56. The most common longhand properties to appear after their corresponding shorthand properties.

As in 2020, the winner was `background-size`, although last year it showed up in 41% of such cases on mobile, and this year was seen in only 15% of such cases.

## Background

Since background longhand properties were at the top of the previous section's chart, we turned our attention to the use of background shorthands and longhands.



Figure 1.57. The most commonly used background properties.

It will come as little surprise that these are used almost universally; if anything, it came as a small surprise that there were any pages that didn't set them. An overwhelming 96% of pages used the `background` shorthand, which goes back to CSS1 in 1996. The same went for the longhand properties of the same age, which were found being applied 85% or more of pages.

That said, the much more recent `background-size` has seen rapid and widespread adoption, appearing in 82% of pages, speaking to its incredible utility to authors. At the other end of the spectrum is `background-origin`, which dropped from 12% usage last year to just 5% this year.

## Margins and paddings

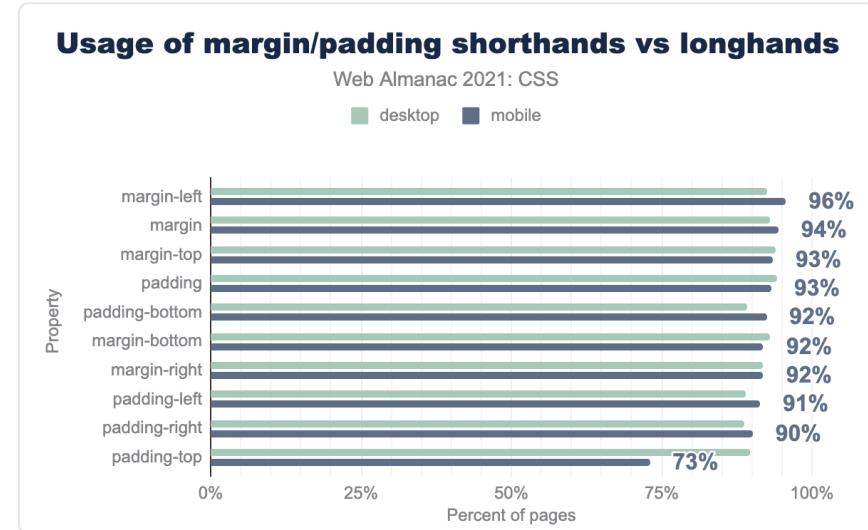


Figure 1.58. The most commonly used margin and padding properties.

Moving down the list, we took a look at margin and padding properties. Much as with backgrounds, it's more a surprise that any pages don't set these properties than that so many do. What interested us this year was that the longhand `margin-left` edged out its shorthand counterpart `margin` to take the top ranking.

## Font



Figure 1.59. The most commonly used font properties

Just as was the case in 2020, the shorthand `font` came in behind all of its common longhand counterparts, with `font-size` leading the way and taking the top spot from last year's winner, `font-weight`.

The also-rans here, `font-variant` and `font-stretch`, have two very different stories. `font-variant` has been around since CSS1, but never really caught on with designers, perhaps because for a long time, the only thing you could do with it was set `small-caps`. Nowadays you can do a lot more with it and downloadable fonts, but authors do not seem to be making use of this capability. Its use dropped significantly this year, down from 43% in 2020 to 23% in 2021.

It's worth taking a little closer look at `font-variant`. While it's used on 23% of mobile pages, the longhand properties that it's now a shorthand for are barely used at all. Here are the actual number of pages found that use not just `font-variant`, but each of its corresponding longhands.

<b>Property</b>	<b>Desktop</b>	<b>Mobile</b>
<code>font-variant</code>	3,098,211	3,641,216
<code>font-variant-numeric</code>	153,932	166,744
<code>font-variant-ligatures</code>	107,211	112,345
<code>font-variant-caps</code>	81,734	86,673
<code>font-variant-east-asian</code>	20,662	20,340
<code>font-variant-position</code>	5,198	5,842
<code>font-variant-alternates</code>	4,876	5,511

Figure 1.60. Number of pages using `font-variant` properties.

Does this mean authors are only using the shorthand, and ignoring the longhands? That probably accounts for a lot of the existing usage, but the steep decline in use of `font-variant` since last year makes us wonder if a common framework or tool dropped `font-variant` from its default styles. Either way, authors may be missing out on a lot of font features that are widely supported.

The other low scoring property, `font-stretch`, is heavily dependent on both font families having wide or narrow faces available and authors choosing (or knowing) to make use of them, so its 5% share (down from 8% last year) comes as little surprise.

## Flexbox



Figure 1.61. The most commonly used Flexbox-related properties.

Some of the Flexbox longhand and shorthand properties have had a turbulent history; for example, the CSS Flexbox specification itself recommends that authors avoid<sup>8</sup> using `flex-grow`, `flex-shrink`, and `flex-basis` and use the `flex` shorthand instead. This ensures that unset properties have sensible values. Unfortunately, this doesn't seem to be bearing out in the wild, where `flex-basis` is used more often on mobile pages than is `flex`, by a margin of more than 10%.

It must be noted that there is a great deal of volatility in these figures as compared to last year's, such as `flex-basis` doubling in usage on mobile while not really shifting on desktop. This could be due to changes in a common framework used in mobile development, or it could be some other factor.

8. <https://drafts.csswg.org/css-flexbox-1/#flex-grow-property>

## Grid



Figure 1.62. The most commonly used Grid-related properties.

The pattern observed in past years is that Grid shorthand properties (`grid-template`, `grid`, etc.) are used far less often than the longhand properties they encompass. In fact, both come in at a staggering 0%, right next to each other in the rankings. The rest of the shorthands are all clustered with them, while longhand properties like `grid-template-rows` and `grid-column` enjoy widespread use. In fact, the only longhand property of any notable usage is `grid-gap`, with 24% usage on mobile Grid pages. It will be interesting to see if the more recent, and generic, `gap` will overtake `grid-gap` in years to come.

## CSS mistakes

Sometimes, one can learn as much from a mistake as from a success. We took the opportunity

to look for not just common errors, but things that looked like they should be correct, but weren't.

## Unrecoverable syntax errors

This year's parsing run, which as in 2020 uses the Rework<sup>9</sup> CSS parser, yielded more heartening numbers. Just 0.94% of desktop pages and 0.55% of mobile pages contained an unrecoverable error—that is, an error so bad, it made parsing the entirety of the stylesheet with Rework impossible. There certainly may have been a much greater number of pages with small, recoverable CSS errors, but the unrecoverable-error figures this year are a great deal lower than last year. This may easily indicate a change in Rework, as opposed to a sudden outbreak of syntax cleanup in the wild.

## Nonexistent properties



Figure 1.63. The most common unknown properties.

One of the things we like to check for is the existence of declarations that are syntactically valid, but use properties that don't actually exist. This doesn't count vendor-prefixed properties, but does include malformed vendor-prefixed properties. Indeed, the most widespread non-existent property we found was `webkit-transition` (which lacks the `-` at

9. <https://github.com/reworkcss/css>

the beginning needed for a proper vendor prefix), appearing on 14% of all pages that contained a nonexistent property. Essentially tied with that was `font-smoothing`, an unprefixed version of `-webkit-font-smoothing` that does not actually exist, nor is it likely to<sup>10</sup> any time soon.

## Longhands before shorthands

In the previous section of this chapter, we looked at which longhand properties were most likely to appear after the corresponding shorthand property (e.g., `background` being followed by `background-size` at some point).



Figure 1.64. The most common shorthand properties to (improperly) appear after any of their corresponding longhand properties.

Doing things the other way around, putting a shorthand *after* a longhand, is a depressingly common mistake, and it happens most often with background properties. In all the cases where a longhand was followed by a corresponding shorthand, a background longhand property was overwritten by the values in the `background` shorthand property.

<sup>10</sup>. <https://developer.mozilla.org/docs/Web/CSS/font-smooth>

## Sass

One of the great advantages of CSS preprocessors is that they can reveal what's missing in CSS itself, and can thus be a guide to how CSS should be extended in the future. This has already happened before, with variables being so popular in preprocessors that CSS eventually added custom properties<sup>11</sup> to its repertoire. Other features of preprocessors, like color modifications and nested selectors, are also finding their way into the base language. This is why we devote a section of this chapter to seeing how developers are using Sass, one of the most popular preprocessors on the web today.

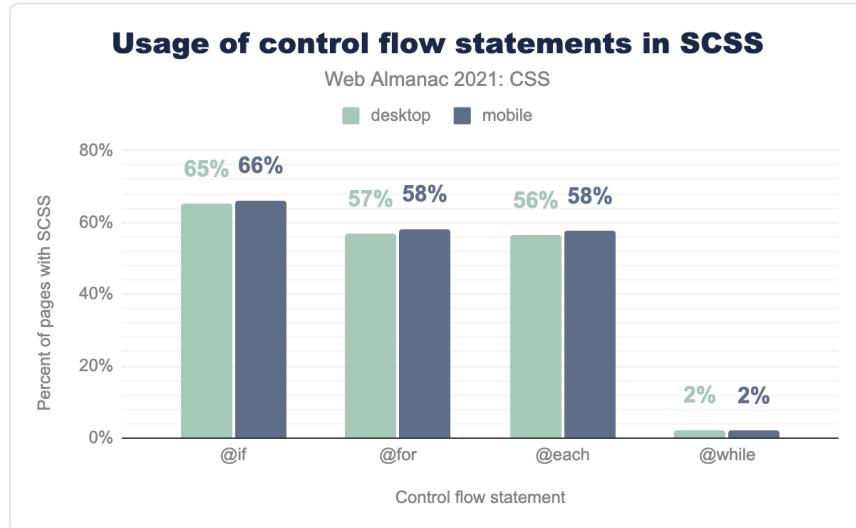


Figure 1.65. The most commonly used Sass function calls.

The Sass functions we found in use largely mirrored those found in the 2020 Web Almanac, albeit with some changes in the specific percentages. When classified by type, we found that 28% of all Sass functions were those that modify colors (e.g., `darken`, `mix`) and a further 6%

11. <https://www.w3.org/TR/css-variables-1/>

were used to read color components (e.g., `alpha`, `blue`).



*Figure 1.66. The most commonly used Sass flow control structures.*

The desire for conditional behavior can be seen in the fact that the `if()` function placed third on the list, at 15% of all Sass functions.

This same desire can be seen even more clearly in the use of Sass's flow control structures, like `@if`. Literally two-thirds of all Sass stylesheets use `@if`, and more than half use `@for` or `@each` (or both). This popular capability was recently added to CSS<sup>12</sup>. By contrast, only 2% use the `@while` structure.

12. <https://drafts.csswg.org/css-conditional-4/#when-rule>



Figure 1.67. The prevalence of rule-nesting in Sass.

Another of Sass’s major draws is the ability to nest rules inside other rules and thus avoid having to write repetitive selector patterns. This capability is under development for native CSS<sup>13</sup>, and our analysis shows why: 87% of all Sass stylesheets use a detectable form of rule nesting. Implicit nesting, which does not require special characters, was not measured.

## Conclusion

In the end, the 2021 Web Almanac tells the story of a technology that’s stable but still evolving. We saw very few instances of major shifts between last year’s Almanac and this year’s—some practices and web features are clearly growing, while others are beginning to fade, but overall, there was a very strong sense of continuity.

Does this mean CSS has become stagnant? Hardly: new layout methods are gaining ground, and major new capabilities are being developed, many of them based on practices worked out in the realm of preprocessors. We would not think to claim that CSS is “solved” or that the best possible practices have already been worked out. As practitioners gain ever more experience,

13. <https://www.w3.org/TR/css-nesting-1/>

changes will come to both CSS the language and CSS the practice. These changes may be gradual rather than sudden, steady rather than disruptive, but this is what we expect in any mature technology.

We look forward to seeing how CSS will grow over the years to come.

## Authors



### Eric A. Meyer

meyerweb <http://meyerweb.com/>

Eric A. Meyer has been a burger flipper, a hardware jockey, a college webmaster, an early blogger, one of the original CSS Samurai<sup>14</sup>, a member of the CSS Working Group<sup>15</sup>, a consultant and trainer, and a Standards Evangelist for Netscape<sup>16</sup>. Currently, he is a Developer Advocate at Igalia<sup>17</sup> and co-founder of An Event Apart<sup>18</sup> with Jeffrey Zeldman<sup>19</sup>. Among other things, Eric co-wrote *Design For Real Life*<sup>20</sup> with Sara Wachter-Boettcher<sup>21</sup> for A Book Apart<sup>22</sup> and *CSS: The Definitive Guide*<sup>23</sup> with Estelle Weyl<sup>24</sup> for O'Reilly<sup>25</sup>, created the first official W3C<sup>26</sup> test suite, and assisted in the creation of microformats<sup>27</sup>.



### Shuvam Manna

@shuvam360 GeekBoySupreme <https://shuvam.xyz>

Shuvam is a designer, doodler<sup>28</sup>, writer<sup>29</sup>, shutterbug<sup>30</sup> and a software tinkerer<sup>31</sup>. He's currently designing at DeepSource<sup>32</sup> and Indie-Hacking, working on Projects such as Doneth and exploring the rough edges of how computers interact with humans.

14. <https://archive.webstandards.org/css/members.html>

15. [https://en.wikipedia.org/wiki/CSS\\_Working\\_group](https://en.wikipedia.org/wiki/CSS_Working_group)

16. <https://en.wikipedia.org/wiki/Netscape>

17. <http://igalia.com/>

18. <https://aeventapart.com/>

19. <http://zeldman.com/>

20. <https://abookapart.com/products/design-for-real-life>

21. <https://sarawb.com>

22. <https://abookapart.com/>

23. <http://meyerweb.com/eric/books/css-tdg/>

24. <http://standardista.com/>

25. <https://oreilly.com/>

26. <http://w3.org/>

27. <http://microformats.org/>

28. <https://www.behance.net/shuvammanna>

29. <https://distortedaura.wordpress.com/>

30. [https://www.instagram.com/the\\_distorted\\_aura/](https://www.instagram.com/the_distorted_aura/)

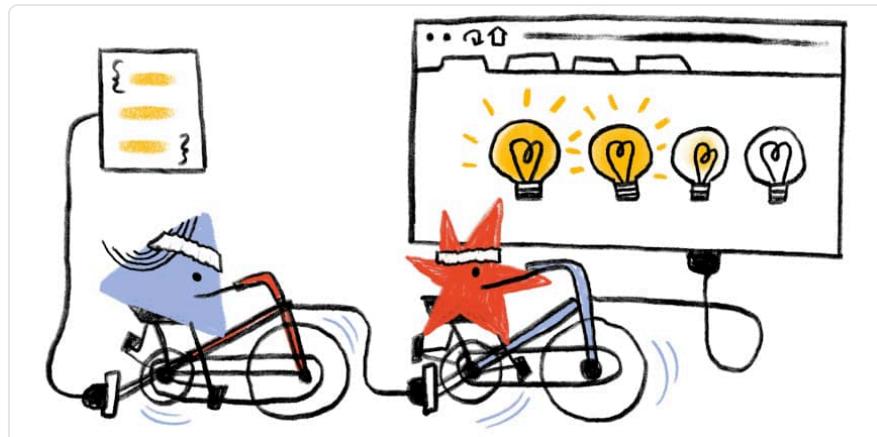
31. <https://github.com/GeekBoySupreme>

32. <https://deepsource.io>



# Part I Chapter 2

# JavaScript



**Written by Nishu Goel**

Reviewed by Manuel Garcia, Minko Gechev, Rick Viscomi, Pankaj Parkar, and Barry Pollard

Analyzed by Pankaj Parkar, Max Ostapenko, and Rick Viscomi

Edited by Rick Viscomi, Pankaj Parkar, and Shaina Hantsis

## Introduction

The speed and consistency at which the JavaScript language has evolved over the past years is tremendous. While in the past it was used primarily on the client side, it has taken a very important and respected place in the world of building services and server-side tools.

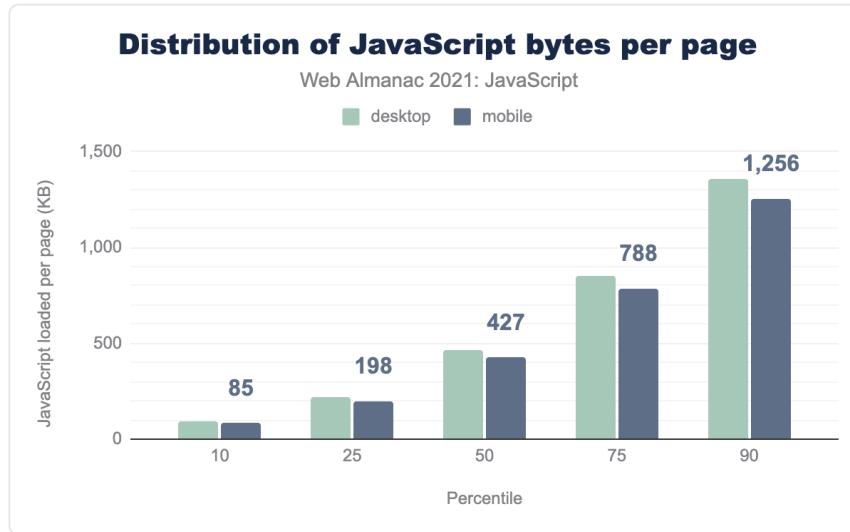
JavaScript has evolved to a point where it is not only possible to create faster applications but also to run servers within browsers<sup>33</sup>.

There is a lot that happens in the browser when rendering the application, from downloading JavaScript to parsing, compiling, and executing it. Let's start with that first step and try to understand how much JavaScript is actually requested by pages.

33. <https://blog.stackblitz.com/posts/introducing-webcontainers/>

## How much JavaScript do we load?

They say, “to measure is the key towards improvement”. To improve the usage of JavaScript in our applications, we need to measure how much of the JavaScript being shipped is actually required. Let’s dig in to understand the distribution of JavaScript bytes per page, considering what a major role it plays in the web setup.



*Figure 2.1. Distribution of the amount of JavaScript loaded per page.*

The 50th percentile (median) mobile page loads 427 KB of JavaScript, whereas the median page loaded on a desktop device sends 463 KB.

Compared to 2019’s results<sup>34</sup>, this shows an increase of 18.4% in the usage of JavaScript for desktop devices and an increase of 18.9% on mobile devices. The trend over time is moving towards using more JavaScript, which could slow down the rendering of an application given the additional CPU work. It’s worth noting that these statistics represent the transferred bytes which could be compressed responses and thus, the actual cost to the CPU could be significantly higher.

Let’s have a look at how much JavaScript is actually required to be loaded on the page.

34. <https://almanac.httparchive.org/en/2019/javascript#fig-2>



Figure 2.2. Distribution of the amount of unused JavaScript bytes on mobile.

According to Lighthouse, the median mobile page loads 155 KB of unused JavaScript. And at the 90th percentile, 598 KB of JavaScript are unused.



Figure 2.3. Distribution of unused and total JavaScript bytes on mobile pages.

# 36.2%

Figure 2.4. Percent unused from the total loaded JavaScript.

To put it another way, 36.2% of JavaScript bytes on the median mobile page go unused. Given the impact JavaScript can have on the Largest Contentful Paint<sup>35</sup> (LCP) of the page, especially for mobile users with limited device capabilities and data plans, this is such a significant figure to be consuming CPU cycles with other important resources just to go to waste. Such wastefulness could be the result of a lot of unused boilerplate code that gets shipped with large frameworks or libraries.

Site owners could reduce the percentage of wasted JavaScript bytes by using Lighthouse to check for unused JavaScript<sup>36</sup> and follow best practices to remove unused code<sup>37</sup>.

## JavaScript requests per page

One of the contributing factors towards slow rendering of the web page could be the requests made on the page, especially when they are blocking requests. It's therefore of interest to look at the number of JavaScript requests made per page on both desktop and mobile devices.

35. <https://web.dev/optimize-lcp/>  
36. <https://web.dev/unused-javascript/>  
37. <https://web.dev/remove-unused-code/>

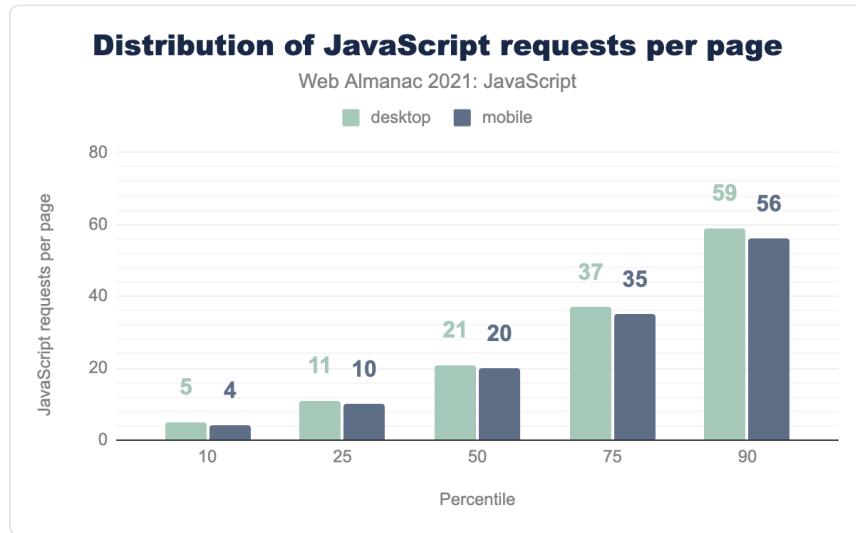


Figure 2.5. Distribution of the number of JavaScript requests per page.

The median desktop page loads 21 JavaScript resources (.js and .mjs requests), going up to 59 resources at the 90th percentile.



Figure 2.6. Distribution of the number of JavaScript requests per page by year.

As compared with last year's results<sup>38</sup>, there has been a marginal increase in the number of JavaScript resources requested in 2021, with the median number of JavaScript resources loaded being 20 for desktop pages and 19 for mobile.

The trend is gradually increasing in the number of JavaScript resources loaded on a page. This would make one wonder if the number should actually increase or decrease considering that fewer JavaScript requests might lead to better performance in some cases but not in others.

This is where the recent advances in the HTTP protocol come in and the idea of reducing the number of JavaScript requests for better performance gets inaccurate. With the introduction of HTTP/2 and HTTP/3, the overhead of HTTP requests has been significantly reduced, so requesting the same resources over more requests is not necessarily a bad thing anymore. To learn more about these protocols, see the HTTP chapter.

## How is JavaScript requested?

JavaScript can be loaded into a page in a number of different ways, and how it is requested can influence the performance of the page.

### `module` and `nomodule`

When loading a website, the browser renders the HTML and requests the appropriate resources. It consumes the polyfills referenced in the code for the effective rendering and functioning of the page. Modern browsers that support newer syntax like arrow functions<sup>39</sup> and async functions<sup>40</sup> do not need loads of polyfills to make things work and therefore, should not have to.

This is when differential loading takes care of things. Specifying the `type="module"` attribute would serve the modern browsers the bundle with modern syntax and fewer polyfills, if any. Similarly, older browsers that lack support for modules will be served the bundle with required polyfills and transpiled code syntax with the `type="nomodule"` attribute. Read more about the usage of module/nomodule<sup>41</sup>.

Let's look at the data to understand the adoption of these attributes.

38. <https://almanac.httparchive.org/en/2020/javascript#request-count>

39. [https://developer.mozilla.org/docs/Web/JavaScript/Reference/Functions/Arrow\\_functions](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Functions/Arrow_functions)

40. [https://developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/async\\_function](https://developer.mozilla.org/docs/Web/JavaScript/Reference/Statements/async_function)

41. [https://developer.mozilla.org/docs/Web/JavaScript/Guide/Modules#applying\\_the\\_module\\_to\\_your\\_html](https://developer.mozilla.org/docs/Web/JavaScript/Guide/Modules#applying_the_module_to_your_html)

Attribute	Desktop	Mobile
<code>module</code>	4.6%	4.3%
<code>nomodule</code>	3.9%	3.9%

Figure 2.7. Distribution of differential loading usage on desktop and mobile clients.

4.6% of desktop pages use the `type="module"` attribute, whereas only 3.9% of mobile pages use `type="nomodule"`. This could be due to the fact that the mobile dataset being much larger contains more “long-tail” websites that might not be using the latest features.

It is important to note that with the end of support for IE 11 browser<sup>42</sup>, differential loading is less applicable because evergreen browsers support modern JavaScript syntax. The Angular framework, for example, removed support for legacy browsers in Angular v13<sup>43</sup>, which was released November 2021.

## async and defer

JavaScript loading could be render-blocking unless it is specified as asynchronous or deferred. This is one of the contributing factors to slow performance, as oftentimes JavaScript (or at least some of it) is needed for the initial render.

However, loading the JavaScript asynchronously or deferred helps in some ways to improve this experience. Both the `async` and `defer` attributes load the scripts asynchronously. The scripts with the `async` attribute are executed irrespective of the order in which they are defined, however, `defer` executes the scripts only after the document is completely parsed, ensuring that their execution will take place in the specified order. Let's look at how many pages actually specify these attributes for the JavaScript requested in the browser.

42. <https://docs.microsoft.com/en-us/lifecycle/announcements/internet-explorer-11-support-end-dates>

43. <https://github.com/angular/angular/issues/41840>

Attribute	Desktop	Mobile
<code>async</code>	89.3%	89.1%
<code>defer</code>	48.1%	47.8%
Both	35.7%	35.6%
Neither	10.3%	10.4%

Figure 2.8. Percent of pages using `async` and `defer`.

There was an anti-pattern observed in last year's results that some websites use both `async` and `defer` attributes on the same script, which falls back to `async` if the browser supports it and using `defer` for IE 8 and IE 9 browsers. This is, however, unnecessary now for most of the sites since `async` takes precedence on all supported browsers and. In turn, this pattern interrupts HTML parsing instead of deferring until the page has loaded. The usage was so frequent that 11.4%<sup>44</sup> of mobile pages were seen with at least one script with `async` and `defer` attributes used together. The root causes<sup>45</sup> were found and an action item was also taken down to remove such usage going forward<sup>46</sup>.

# 35.6%

Figure 2.9. Percent of mobile pages on which the `async` and `defer` attributes are set on the same script.

This year, we found that 35.6% of mobile pages use the `async` and `defer` attributes together. The large discrepancy from last year is due to a methodological improvement to measure attribute usage at runtime, rather than parsing the static contents of the initial HTML. This difference shows that many pages update these attributes dynamically after the document has already been loaded. For example, one website was found to include the following script:

```
<!-- Piwik -->
<script type="text/javascript">
(function() {
    var d=document, g=d.createElement('script'),
```

44. <https://almanac.httparchive.org/en/2020/javascript#how-do-we-load-our-javascript>

45. [https://twitter.com/rick\\_viscomi/status/1331735748060524551?s=20](https://twitter.com/rick_viscomi/status/1331735748060524551?s=20)

```
s=d.getElementsByTagName('script')[0];
  g.type='text/javascript'; g.async=true; g.defer=true;
  g.src=u+'piwik.js'; s.parentNode.insertBefore(g,s);
})();
</script>
<!-- End Piwik Code -->
```

So, what is Piwik? According to its Wikipedia entry:

*Matomo, formerly Piwik, is a free and open source web analytics application developed by a team of international developers, that runs on a PHP/MySQL web server. It tracks online visits to one or more websites and displays reports on these visits for analysis. As of June 2018, Matomo was used by over 1,455,000 websites, or 1.3% of all websites with known traffic analysis tools...*

— Matomo (software) on Wikipedia<sup>47</sup>

This information strongly suggests that much of the increase we observed may be due to similar marketing and analytics providers that dynamically inject these `async` and `defer` scripts into the page later than had been previously detected.

# 2.6%

Figure 2.10. Percent of scripts using the `async` and `defer` attribute together.

Even though a large percentage of pages use this anti-pattern, it turns out that only 2.6% of all scripts use both `async` and `defer` on the same script element.

## First-party vs third-party

Recall from the How much JavaScript do we load section that the median number of JavaScript requests on mobile pages is 20. In this section, we'll take a look at the breakdown of first and third-party JavaScript requests.

47. [https://en.wikipedia.org/wiki/Matomo\\_\(software\)](https://en.wikipedia.org/wiki/Matomo_(software))

## Distribution of JavaScript requests by host

Web Almanac 2021: JavaScript (mobile)




Figure 2.11. Distribution of the number of JavaScript requests per mobile page by host.

The median mobile page requests 10 third-party resources and 9 first-party requests. This difference increases as we move up to the 90th percentile as 33 requests on mobile pages are first-party but the number goes up to 34 for third-party requests for the mobile pages. Clearly, the number of third-party resources requested is always one step ahead of the first-party ones.

## Distribution of JavaScript requests by host

Web Almanac 2021: JavaScript (desktop)

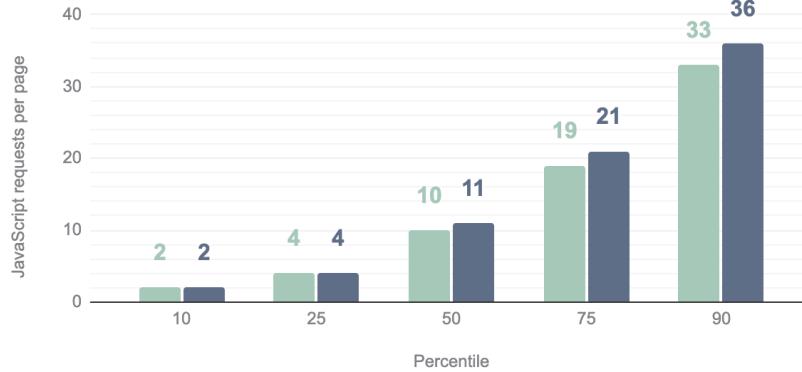



Figure 2.12. Distribution of the number of JavaScript requests per desktop page by host.

The median desktop page requests 11 third-party resources, compared to 10 first-party requests. Irrespective of the performance and reliability risks<sup>48</sup> that third-party resources may bring, both desktop and mobile pages consistently seem to favor third-party scripts. This effect could be due to the useful interactivity features<sup>49</sup> that third-party scripts give to the web.

Nevertheless, site owners must ensure that their third-party scripts are loaded performantly<sup>50</sup>. Harry Roberts<sup>51</sup> advocates for going a step further and stress testing third-parties<sup>52</sup> for performance and resilience.

## preload and prefetch

As a page is rendered, the browser downloads the given resources and prioritizes the download of some resources the browser uses over others using resource hints. The `preload` hint tells the browser to download the resource with a higher priority as it will be required on the current page. The `prefetch` hint, however, tells the browser that the resource could be required after some time (useful for future navigation) and it'd better to fetch it when the browser has the capacity to do so and make it available as soon as it is required. Learn more about how these features are used in the Resource Hints chapter.

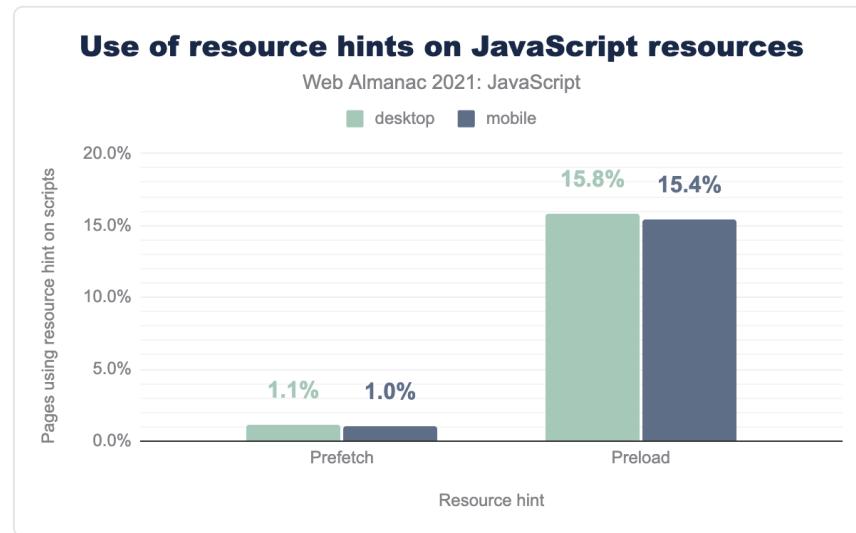


Figure 2.13. Use of resource hints on JavaScript resources.

48. <https://css-tricks.com/potential-dangers-of-third-party-javascript/>  
 49. <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/adding-interactivity-with-javascript>  
 50. <https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/loading-third-party-javascript>  
 51. <https://twitter.com/csswizardry>  
 52. <https://csswizardry.com/2017/07/performance-and-resilience-stress-testing-third-parties/>

`preload` hints are used to load JavaScript on 15.4% of mobile pages, whereas only 1.0% of mobile pages use the `prefetch` hint. 15.8% and 1.1% of desktop pages use these resource hints to load JavaScript resources, respectively.

It would also be useful to see how many `preload` and `prefetch` hints are used per page, as that affects the impact of these hints. For example, if there are five resources to be loaded on the render and all five use the `preload` hint, the browser would try to prioritize every resource, which would effectively work as if no `preload` hint was used at all.



Figure 2.14. Distribution of preload hints for JavaScript resources per page.



Figure 2.15. Distribution of prefetch hints for JavaScript resources per page.

The median desktop page loads one JavaScript resource with the `preload` hint and two JavaScript resources with the `prefetch` hint.

Hint	2020	2021
<code>preload</code>	1	1
<code>prefetch</code>	3	2

Figure 2.16. Year-over-year comparison of the median number of `preload` and `prefetch` hints for JavaScript resources per mobile page.

While the median number of `preload` hints per mobile page has stayed the same, the number of `prefetch` hints has decreased from three to two per page. Note that at the median, these results are identical for both mobile and desktop pages.

## How is JavaScript delivered?

JavaScript resources can be loaded more efficiently over the network with compression and minification. In this section, we'll explore the usage of both techniques to better understand the extent to which they're being utilized effectively.

## Compression

Compression is the process of reducing the file size of a resource as it gets transferred over the network. This can be an effective way to improve the download times of JavaScript resources, which are highly compressible. For example, the `almanac.js` script loaded on this page is 28 KB, but only 9 KB over the wire thanks to compression. You can learn more about the ways resources are compressed across the web in the Compression chapter.



*Figure 2.17. Adoption of the methods for compressing JavaScript resources.*

Most JavaScript resources are either compressed using Gzip<sup>53</sup>, Brotli<sup>54</sup> (br), or not compressed at all (not set). 55.4% of mobile JavaScript resources use Gzip, whereas 30.8% of resources are compressed with Brotli.

Interestingly, compared to the state of JavaScript compression in 2019<sup>55</sup>, Gzip has gone down by almost 10 percentage points and Brotli has increased by 16 percentage points. The trend illustrates the shift to focus on smaller size files with higher levels of compression that Brotli provides as compared to Gzip.

To help explain this change, we analyzed the compression methods of first and third-party resources.

53. <https://www.gnu.org/software/gzip/manual/gzip.html>

54. <https://github.com/google/brotli>

55. <https://almanac.httparchive.org/en/2019/javascript#fg-10>

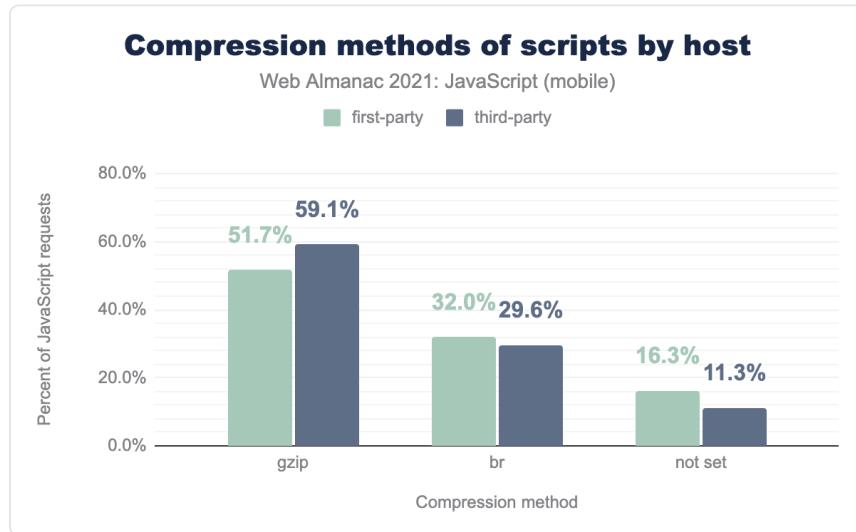


Figure 2.18. Adoption of the methods for compressing first and third-party JavaScript resources on mobile pages.

59.1% of third-party scripts on mobile pages are gzipped and 29.6% are compressed with Brotli. Looking at first-party scripts, these are 51.7% with Gzip compression but only 32.0% with Brotli. There are still 11.3% of third-party scripts that do not have any compression method defined.



Figure 2.19. Uncompressed resources for first party vs third party.

90% of uncompressed third-party JavaScript resources are less than 5 KB, though first-party requests trail a bit. This may help explain why so many JavaScript resources go uncompressed. Due to the diminishing returns of compressing small resources, a small script may cost more in terms of the resource consumption of server-side compression and client-side decompression than the performance benefits of saving a few bytes over the network.

## Minification

While compression only changes the transfer size of JavaScript resources over the network, minification actually makes the code itself smaller and more efficient. This not only helps to reduce the load time of the script but also the amount of time the client spends parsing the script.

The unminified JavaScript<sup>56</sup> Lighthouse audit highlights the opportunities of minification.

56. <https://web.dev/unminified-javascript/>

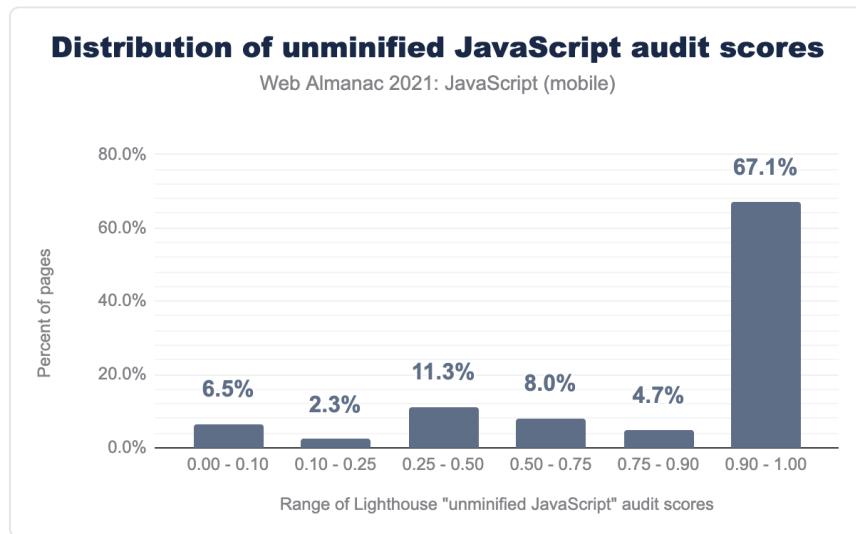


Figure 2.20. Distribution of unminified JavaScript audit scores.

Here, 0.00 represents the worst score whereas 1.00 represents the best score. 67.1% of mobile pages have an audit score between 0.9 and 1.0. That means there are still more than 30% of pages that have an unminified JavaScript score worse than 0.9 and could make better use of code minification. Compared to the results from the 2020 edition<sup>57</sup>, the percent of mobile pages with an “unminified JS” score between 0.9 and 1.0 fell by 10 points.

To understand the reason for the worse scores this year, let’s dive deeper to look at how many bytes per page are unminified.

57. <https://almanac.httparchive.org/en/2020/javascript#fig-16>

## Unminified JavaScript bytes per page

Web Almanac 2020: JavaScript (mobile)

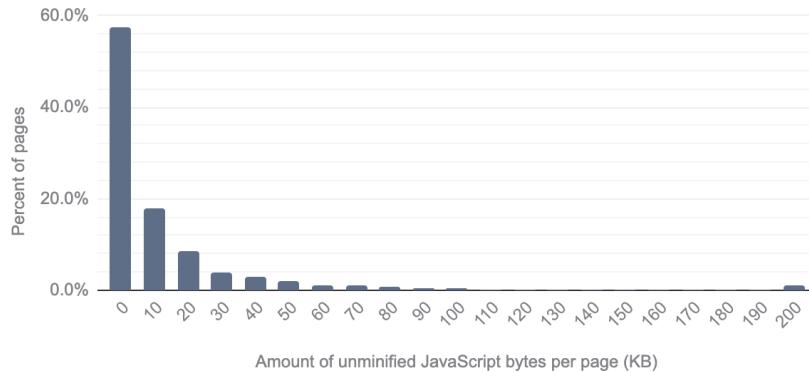


Figure 2.21. Distribution of the amount of unminified JavaScript per page, in KB.

57.4% of mobile pages have 0 KB of unminified JavaScript as reported by the Lighthouse audit. 17.9% of mobile pages have between 0 and 10 KB of unminified JavaScript. The rest of the pages have an increasing number of unminified JavaScript bytes and correspond to those having poor “unminified JavaScript” audit scores in the previous chart.

## Average distribution of unminified JS bytes

Web Almanac 2021: JavaScript (mobile)

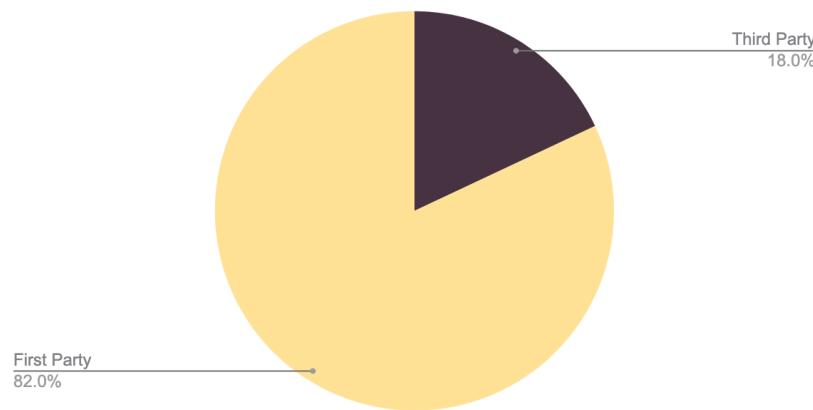


Figure 2.22. Average distribution of unminified JavaScript bytes.

When we segmented the unminified JavaScript resources by host, we found that 82.0% of the average mobile page's unminified JavaScript bytes actually come from first-party scripts.

## Source maps

Source maps<sup>58</sup> are hints sent along with JavaScript resources that allow the browser to map the minified resource back to their source code. This is especially helpful to web developers for debugging in a production environment.



0.1%

Figure 2.23. Percent of mobile pages that use the `SourceMap` header.

Only 0.1% of mobile pages use the `SourceMap` response header on script resources. One reason for this extremely small percentage could be that not many sites choose to put their original source code in production through the source map.



98.0%

Figure 2.24. Percent of JavaScript resources on mobile pages using the `SourceMap` header that are first-party resources.

98.0% of the `SourceMap` usage on JavaScript resources can be attributed to first-parties. Only 2.0% of scripts with the header on mobile pages are third-party resources.

## Libraries and frameworks

The usage of JavaScript seems to have increased tremendously over the years, with the adoption of many new libraries and frameworks all promising their own unique improvements to the developer and user experiences. They have become so prevalent that the term *framework fatigue* was coined to describe developers' struggle just to keep up. In this section, we'll look at the popularity of the JavaScript libraries and frameworks in use on the web today.

---

58. [https://developer.mozilla.org/docs/Tools/Debugger/How\\_to/Use\\_a\\_source\\_map](https://developer.mozilla.org/docs/Tools/Debugger/How_to/Use_a_source_map)

## Libraries usage

To understand the usage of libraries and frameworks, HTTP Archive uses Wappalyzer to detect the technologies used on a page.

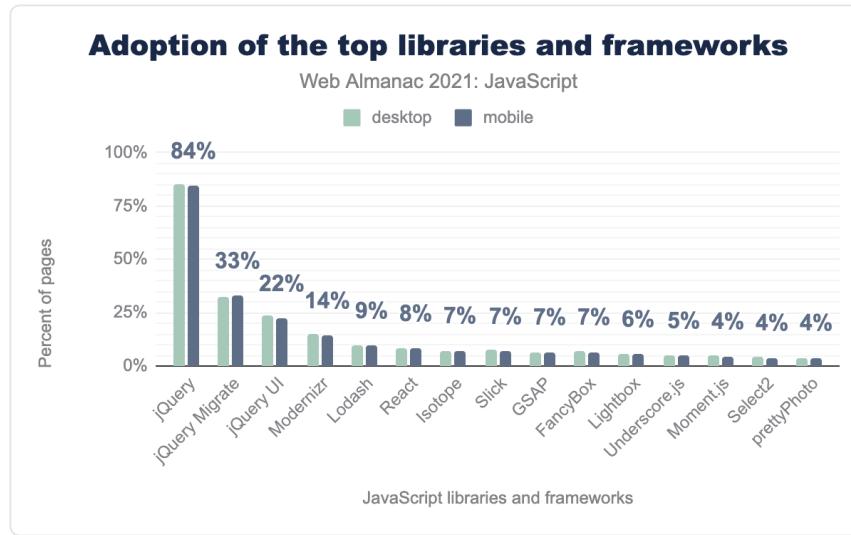


Figure 2.25. Usage of JavaScript libraries and frameworks.

jQuery remains the most popular library, used by a staggering 84% of mobile pages. React usage has jumped from 4% to 8% since last year, which is a significant increase. React's increase may be partially due to recent detection improvements<sup>59</sup> to Wappalyzer, and may not necessarily reflect the actual change in adoption. It's also worth noting that Isotope, which uses jQuery, is found on 7% of pages, leading to RequireJS falling out of the top spots on just 2% of pages.

You might wonder why jQuery is still so dominant in 2021. There are two main reasons for this. First, as highlighted over the previous years<sup>60</sup>, most WordPress<sup>61</sup> sites use jQuery. Given that WordPress is used on nearly a third of all websites, according to the CMS chapter, this accounts for a huge proportion of jQuery adoption. Second, several of the other top-used JavaScript libraries still rely on jQuery in some way under the hood, contributing to indirect adoption of the library.

59. <https://github.com/AlisalO/wappalyzer/issues/2450>

60. <https://almanac.httparchive.org/en/2019/javascript/#open-source-libraries-and-frameworks>

# 3.5.1

*Figure 2.26. The most popular version of jQuery.*

The most popular version of jQuery is 3.5.1, which is used by 21.3% of mobile pages. The next most popular version of jQuery is 1.12.4, at 14.4% of mobile pages. The leap to version 3.0 can be explained by a change to WordPress core<sup>62</sup> in 2020, which upgraded the default version of jQuery from 1.12.4 to 3.5.1.

## Libraries used together

Now let's look at how the popular frameworks and libraries are used together on the same page.

Frameworks and libraries	Desktop	Mobile
jQuery	16.8%	17.4%
jQuery, jQuery Migrate	8.4%	8.7%
jQuery, jQuery UI	4.0%	3.7%
jQuery, jQuery Migrate, jQuery UI	2.6%	2.5%
Modernizr, jQuery	1.6%	1.6%
FancyBox, jQuery	1.1%	1.1%
Slick, jQuery	1.2%	1.1%
Lightbox, jQuery	1.1%	0.8%
React, jQuery, jQuery Migrate	0.9%	0.9%
Modernizr, jQuery, jQuery Migrate	0.8%	0.9%

*Figure 2.27. Top combinations of JavaScript frameworks and libraries used together.*

The most widely-used combination of JavaScript libraries and frameworks doesn't actually consist of multiple libraries at all! When used by itself, jQuery is found on 17.4% of mobile pages. The next most popular combination is jQuery and jQuery Migrate, which is used on 8.7%

62. <https://wptavern.com/major-jquery-changes-on-the-way-for-wordpress-5-5-and-beyond>

of mobile pages. In fact, all of the top 10 library and framework combinations include jQuery.

## Security vulnerabilities

Using JavaScript libraries can come with its own benefits and drawbacks. When using these libraries, one drawback is that older versions may include security risks like Cross Site Scripting<sup>63</sup> (XSS). Lighthouse detects the JavaScript libraries used on a page and fails the audit if their version has any known vulnerabilities in the open-source Snyk vulnerability database<sup>64</sup>.

A large, bold, blue percentage graphic showing 63.9%.

Figure 2.28. Percentage of mobile pages with libraries having a security vulnerability.

63.9% of mobile pages use a JavaScript library or framework with a known security vulnerability. For context, this number has come down from 83.5% since last year<sup>65</sup>.

---

63. <https://owasp.org/www-community/attacks/xss/>  
64. <https://snyk.io/vuln?type=npm>  
65. <https://almanac.httparchive.org/en/2020/javascript#fg-30>

<b>Library or framework</b>	<b>Percent of pages</b>
jQuery	57.6%
Bootstrap	12.2%
jQuery UI	10.5%
Underscore	6.4%
Lo-Dash	3.1%
Moment.js	2.3%
GreenSock JS	1.8%
Handlebars	1.3%
AngularJS	1.0%
Mustache	0.7%
jQuery Mobile	0.5%
Dojo	0.5%
Angular	0.4%
Vue	0.2%
Knockout	0.2%
Highcharts	0.1%
Next.js	0.0%
React	0.0%

Figure 2.29. The percent of mobile pages found to contain a vulnerable version of a JavaScript library or framework.

When we segment the percent of mobile pages by library and framework, we can see that jQuery is largely responsible for the decrease in vulnerabilities. This year JavaScript vulnerabilities were found on 57.6% of pages with jQuery, compared to 80.9% last year<sup>66</sup>. As predicted<sup>67</sup> by Tim Kadlec<sup>68</sup> in the 2020 edition of this chapter, “if we can get folks to migrate away from those outdated, vulnerable versions of jQuery, we would see the number of sites with known

66. <https://almanac.httparchive.org/en/2020/javascript#fig-31>  
 67. <https://almanac.httparchive.org/en/2020/javascript#fig-31>  
 68. <https://almanac.httparchive.org/en/2020/contributors#tkadlec>

vulnerabilities plummet". And that's exactly what happened; WordPress migrated from jQuery version 1.12.4 to the more secure version 3.5.1, contributing to a 20 point drop in the percent of pages with known JavaScript vulnerabilities.

## How do we use JavaScript?

Now that we've looked at how we get the JavaScript, what are we using it for?

### AJAX

One way that JavaScript is used is to communicate with servers to asynchronously receive information in various formats. *Asynchronous JavaScript and XML* (AJAX) is typically used to send and receive data, and it supports more than just XML, including JSON, HTML, and text formats.

With multiple ways to send and receive data on the web, let's look at how many asynchronous requests are sent per page.



Figure 2.30. Distribution of the number of asynchronous requests made per page.

The median mobile page makes 4 asynchronous requests. If we look at the long tail, the largest number of asynchronous requests on desktop pages is 623, which is eclipsed by the biggest mobile page, which makes 867 asynchronous requests!

An alternative to the asynchronous AJAX requests are the synchronous requests. Rather than passing a request to a callback, they block the main thread until the request completes.

However, this practice is discouraged<sup>69</sup> due to the potential for poor performance and user experiences, and many browsers already warn about such usage. It would be intriguing to see how many pages still use synchronous AJAX requests.



Figure 2.31. Usage of synchronous and asynchronous AJAX requests.

2.5% of mobile pages use the deprecated synchronous AJAX requests. To put this into perspective, let's look at the trend by comparing the results with the last two years.

69. [https://developer.mozilla.org/docs/Web/API/XMLHttpRequest/Synchronous\\_and\\_Asynchronous\\_Requests#synchronous\\_request](https://developer.mozilla.org/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Requests#synchronous_request)



Figure 2.32. Usage of synchronous and asynchronous AJAX requests over years.

We see that there is a clear increase in the usage of asynchronous AJAX requests. However, there isn't a significant decline in the usage of synchronous AJAX requests.

Knowing the number of AJAX requests per page now, we'd also be interested in knowing the most commonly used APIs to request the data from the server.

We can broadly classify these AJAX requests into three different APIs and dig in to see how they're used. The core APIs `XMLHttpRequest` (XHR), `Fetch`, and `Beacon` are used commonly across websites with XHR being used primarily, however `Fetch` is gaining popularity and growing rapidly while `Beacon` has low usage.



Figure 2.33. Distribution of the number of XMLHttpRequest requests per page.

The median mobile page makes 2 XHR requests, but at 90th percentile, makes 6 XHR requests.

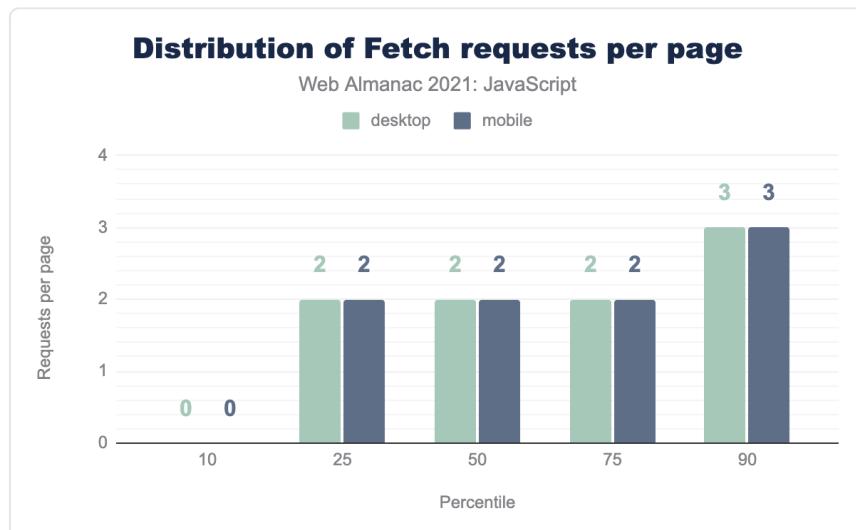


Figure 2.34. Distribution of the number of `Fetch` requests per page.

In the case of the usage of the `Fetch` API, the median mobile page makes 2 requests, and in the long tail, reaches 3 requests. This API is becoming the standard XHR way of making

requests, due in part to its cleaner approach and less boilerplate code. There may also be performance benefits<sup>70</sup> to `Fetch` over the traditional XHR approach, due to the way browsers can decode large JSON payloads off the main thread.



Figure 2.35. Distribution of the number of `Beacon` requests per page.

`Beacon` usage is almost non-existent, with 0 requests per page until the 90th percentile, at which there's only one request per page. One possible explanation for this low adoption could be that `Beacon` is typically used for sending analytics data, especially when one wants to ensure that the request is sent even if the page might unload soon. This is, however, not guaranteed when using XHR. A good experiment for the future would be to see if some statistics could be collected around any pages using XHR for analytics data, session data, etc.

It would be interesting to also compare the adoption of XHR and `Fetch` over time.

70. <https://gomakethings.com/the-fetch-api-performance-vs-xhr-in-vanilla-js/>



Figure 2.36. Adoption of AJAX APIs by year.

For both `Fetch` and `XHR`, the usage has increased significantly over the years. `Fetch` usage on mobile pages is up 4 points and `XHR` is up 19 points. The gradual increase of `Fetch` adoption seems to point towards a trend of cleaner requests and better response handling.

## Web Components and the shadow DOM

With the web becoming componentized<sup>71</sup>, a developer building a single page application may think about a user view as a set of components. This is not only for the sake of developers building dedicated components for each feature, but also to maximize component reusability. It could be in the same app on a different view or in a completely different application. Such use cases lead to the usage of custom elements and web components in applications.

It would be justified to say that with many JavaScript frameworks gaining popularity, the idea of reusability and building dedicated feature-based components has been adopted more widely. This feeds our curiosity to look into the adoption of custom elements, shadow DOM, template elements.

Custom Elements<sup>72</sup> are customized elements built on top of the `HTMLElement` API. Browsers provide a `customElements` API that allows developers to define an element and register it with the browser as a custom element.

71. [https://developer.mozilla.org/docs/Web/Web\\_Components](https://developer.mozilla.org/docs/Web/Web_Components)

72. <https://developers.google.com/web/fundamentals/web-components/customelements>

# 3.0%

Figure 2.37. Percent of desktop pages using custom elements.

3.0% of mobile pages use custom elements for one or more parts of the web page.

# 0.4%

Figure 2.38. Percent of pages using Shadow DOM.

Shadow DOM allows you to create a dedicated subtree in the DOM for the custom element introduced to the browser. It ensures the styles and nodes inside the element are not accessible outside the element.

0.4% of mobile pages use shadow DOM specification of web components to ensure a scoped subtree for the element.

# <0.1%

Figure 2.39. Percent of pages using `template` elements.

A `template` element is very useful when there is a pattern in the markup which could be reused. The contents of `template` elements render only when referenced by JavaScript.

Templates work well when dealing with web components, as the content that is not yet referenced by JavaScript is then appended to a shadow root using the shadow DOM.

Fewer than 0.1% of web pages have adopted the use of templates. Although templates are well supported<sup>73</sup> in browsers, there is still a very low percentage of pages using them.

## Conclusion

The numbers that we have seen throughout the chapter have brought us to an understanding of

73. <https://caniuse.com/template>

how vast the JavaScript usage is and how it's evolving over time. The JavaScript ecosystem has been growing with the focus towards making the web more performant and secure for users, with newer features and APIs that make the developer experience easier and more productive.

We saw how so many features that improve rendering and resource loading performance could be more widely utilized to provide users with faster experiences. As a developer, you can start by adopting these new web platform features. However, make sure to use them wisely and ensure that they actually improve performance, as some APIs can cause harm through misuse, as we saw with `async` and `defer` attributes on the same script.

Making appropriate use of the powerful APIs that we now have access to is what it will take to see these numbers improve further in the coming years. Let's continue to do so.

## Author



Nishu Goel

🐦 @TheNishuGoel 🌐 NishuGoel 🌐 <http://unravelweb.dev/>

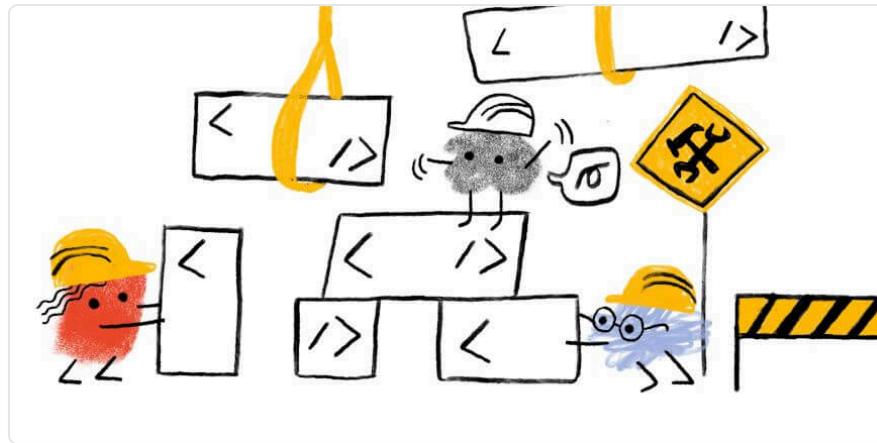
Nishu Goel is an engineer at Web DataWorks<sup>74</sup>. She is a Google Developer Expert for Web Technologies and Angular, Microsoft MVP for Developer Technologies, and the author of Step by Step Guide Angular Routing (BPB, 2019) and A Hands-on Guide to Angular (Educative, 2021). Find her writings at [unravelweb.dev](http://unravelweb.dev/)<sup>75</sup>.

74. <http://webdataworks.io/>  
75. <https://unravelweb.dev/>



# Part I Chapter 3

# Markup



Written by Alex Lakatos

Reviewed by Jens Oliver Meiert, Brian Kardell, Shaina Hantsis, Barry Pollard, and Rick Viscomi

Analyzed by Kevin Farrugia

Edited by Rick Viscomi

## Introduction

Have you ever wondered what happens when you try to visit a web site? After you enter the URL in the address bar of your browser, one of the first things that happens is that a HTML file is downloaded and parsed. You could say that markup is the foundation of the Web. We've dedicated this chapter to looking at some of the bricks that make the web stand today.

We've drawn on the data analyzed for the past three years to try to come up with a few questions around the future of markup, the trends emerging over the years, and the adoption rate of new standards. We've also shared the data in the hopes that you'll dig deeper into it, and interpret it in a way that we haven't.

*In the Markup chapter, we focus on HTML. While we briefly touch on other markup languages (like SVG or MathML) or other topics in the Web Almanac, those are covered in more detail in their own dedicated chapters. Because the markup is the gateway into the web, it was extremely hard not to dedicate a whole chapter to it.*

## General

We'll start with some of the more general aspects of a markup document: things like document types, document sizes, document language, and compression.

### Doctypes

Ever wondered why all pages start with `<!DOCTYPE html>` or something similar, even in 2021? Doctypes are required because they tell the browsers not to switch into "quirks mode"<sup>76</sup> when rendering a page, and instead, they should make a best-effort attempt to follow the HTML spec.

This year, 97.4% of pages had a doctype, slightly up from last year's 96.8%. Looking at the past couple of years, the doctype percentage has increased steadily by half a percentage point every year. In an ideal world, 100% of web pages would have a doctype—at this rate, we'll live in an ideal world by 2027!

In terms of popularity, HTML5, better known as `<!DOCTYPE html>` is still the most popular doctype, with 88.8% of mobile pages using it.

<b>Doctype</b>	<b>Desktop</b>	<b>Mobile</b>
HTML ("HTML5")	87.0%	88.8%
XHTML 1.0 Transitional	5.7%	4.6%
XHTML 1.0 Strict	1.4%	1.3%
HTML 4.01 Transitional	0.9%	0.7%
HTML 4.01 Transitional (quirky <sup>77</sup> )	0.5%	0.5%

Figure 3.1. Most popular doctypes.

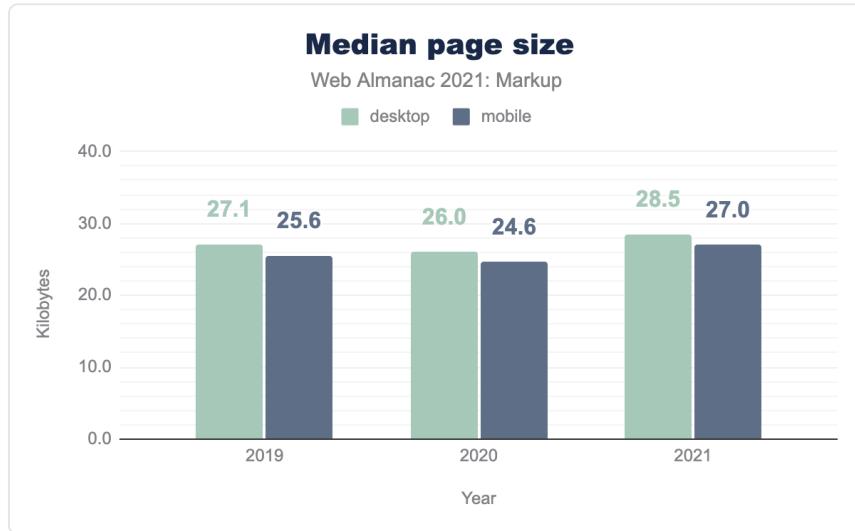
The surprising part is that, almost 20 years later<sup>78</sup>, XHTML is still a considerable part of the web, with 8% of pages still using it on desktop and a little under 7% on mobile.

### Document size

In a mobile world, where every byte of data has a cost associated with it, document sizes for

76. [https://developer.mozilla.org/docs/Web/HTML/Quirks\\_Mode\\_and\\_Standards\\_Mode](https://developer.mozilla.org/docs/Web/HTML/Quirks_Mode_and_Standards_Mode)  
 77. <https://isivonen.fi/doctype/#xml>  
 78. <https://en.wikipedia.org/wiki/XHTML>

mobile websites are becoming increasingly more important. It is also increasingly bigger, by the looks of it. This year, the median mobile page had 27 KB of HTML, up 2 KB from last year. On the desktop side, the median page had 29 KB of HTML.



*Figure 3.2. The median page size year-over-year.*

The interesting points were:

- The median page sizes in 2020 were shrinking when compared to 2019. Looking at the figure above, we've had a slight increase this year, after the dip in 2020.
- The biggest HTML documents for both desktop and mobile have shed a whopping 20 MB each this year, with the biggest ones being 45 MB on desktop and 21 MB on mobile.

## Compression

With document sizes increasing, we also looked at compression this year. We felt the document size relates closely to the level of compression used when transferring it over the wire.

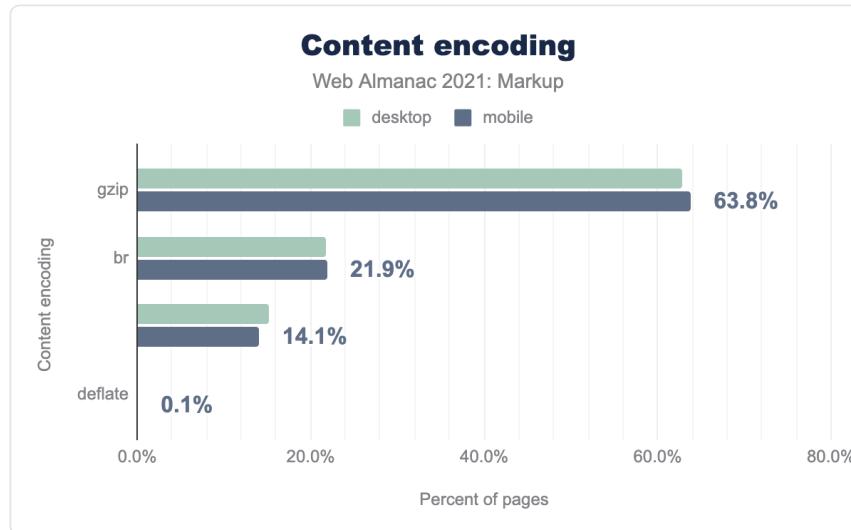


Figure 3.3. Adoption of content encoding schemes.

Out of the 6 million desktop pages scanned, an overwhelming 84.4% were compressed with either gzip (62.7%) or Brotli (21.7%) compression. For mobile pages, the numbers are very similar, 85.6% were compressed with either gzip (63.7%) or Brotli (21.9%) compression. The slight variation in percentages for mobile and desktop is not surprising, as they comprise of different URLs, and the Mobile data set is a lot larger.

Compression is important as, particularly in a mobile world, every byte of data has a cost associated with it. You can learn more about the states of content encoding and the mobile web in the Compression and Mobile Web chapters.

## Document language

We've encountered 3,598 unique instances of the `lang` attribute on the `html` element. Because there are 7,139 spoken languages<sup>79</sup> at the time of writing this chapter, it made us think not all of them were represented. When we factored in the script and region subtags<sup>80</sup>, even fewer remained.

79. <https://www.ethnologue.com/guides/how-many-languages>  
 80. [https://developer.mozilla.org/docs/Web/HTML/Global\\_attributes/lang#language\\_tag\\_syntax](https://developer.mozilla.org/docs/Web/HTML/Global_attributes/lang#language_tag_syntax)

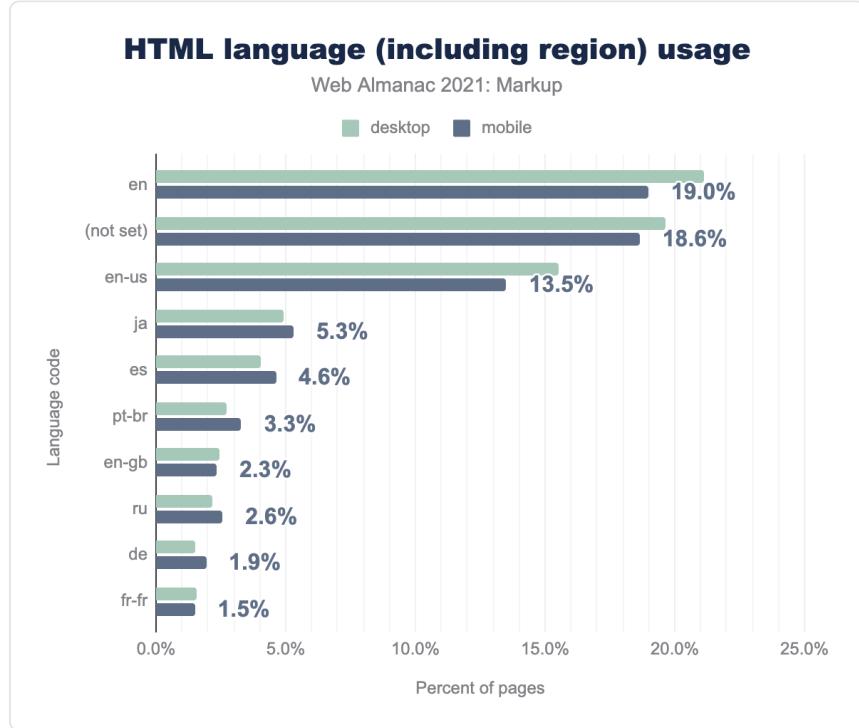


Figure 3.4. Adoption of the most popular HTML language codes, including region.

Out of the pages scanned, 19.6% on desktop, and 18.6% on mobile, specified no `lang` attribute, even though the Web Content Accessibility Guidelines (WCAG<sup>81</sup>) requires that a page language is defined and “programmatically accessible”. Languages can be specified in different ways, including an `xml:lang` element, which we didn’t check for, so there might still be hope for some of the pages scanned.

81. <https://www.w3.org/TR/UNDERSTANDING-WCAG20/meaning-doc-lang-id.html>



Figure 3.5. Adoption of the most popular HTML language codes, not including region.

While we looked at the top 10 normalized languages in the set, some interesting trends emerged:

- Mobile has a lower relative percentage of English websites. We're not sure why that is the case, we've been discussing the cause as a team. It's possible that some people only use mobile phones to access the web, so that would diversify the mobile set's language landscape. This author believes a lot of the mobile pages are intended to be used on the go and hence are local.
- While Spanish has a lot more region and subscript options than Japanese, it was a tight contest for the second most popular language.
- There is an inverse correlation between the difference in empty attributes for desktop and mobile and English.

## Comments

# 88%

Figure 3.6. Pages with at least one comment in HTML.

Most production build tools have an option to remove comments, but we've found a majority of the pages we've analyzed, 88%, had at least one comment.

While comments are generally encouraged in code, a particular type of comment, conditional comments, were used in web pages to render markup for particular browsers.

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

Microsoft dropped support for conditional comments in IE 10. Still, 41% of the pages had at least one conditional comment present. Aside from the possibility that these are very old websites, we could only assume they are using some sort of variation of polyfilling framework for older browsers.

## SVG use

# 46.4%

Figure 3.7. Pages with at least one SVG element in HTML.

This year, we wanted to take a look at SVG usage. With popular icon libraries using more and more SVG, favicon support improving, and SVG images being on the rise in animations, it's no surprise that 46.4% of web pages had some sort of SVG on them. 37.2% had a SVG element, 20.0% on desktop and 18.4% on mobile were using SVG images, and a negligible amount had either SVG embeds, objects, or iframes in them.

SGVs have more use cases when compared to the style element, but in terms of popularity, the numbers are comparable. SVG sits just outside the top 20 in terms of element popularity on a page.

## Elements

Elements are the DNA of a HTML document. We wanted to analyze the cells that make up the living organism that is a web page. What are the most popular, the most likely to be present, and the obsolete elements on most pages?

### Element diversity

There are 112 elements<sup>82</sup> currently defined and in use (excepting SVG and MathML), with another 28 being deprecated<sup>83</sup> or obsolete. We wanted to see how many of them were actually used on a page, and how likely a web of `div`s was.



Figure 3.8. Distribution of the number of distinct types of elements per page.

No need to panic, the web isn't all made up of `div`s. The median mobile page uses 31 different elements and has a total of 616 elements.

82. <https://html.spec.whatwg.org/multipage/indices.html#elements-3>

83. [https://developer.mozilla.org/docs/Web/HTML/Element#obsolete\\_and\\_deprecated\\_elements](https://developer.mozilla.org/docs/Web/HTML/Element#obsolete_and_deprecated_elements)



*Figure 3.9. Distribution of the number elements per page.*

While the median page had 666 elements on desktop, and 616 on mobile, the top 10% of all pages had closer to triple that number, 1,727 for mobile and 1,902 for desktop.

## Top elements

Every year since 2019, the Markup chapter of the Web Almanac has featured the most frequently used elements in reference to Ian Hickson's work in 2005<sup>84</sup>. This author couldn't break with tradition, so we had a look at the data again.

84. <https://web.archive.org/web/20060203031713/http://code.google.com/webstats/2005-12/elements.html>

2005	2019	2020	2021
<i>title</i>	<i>div</i>	<i>div</i>	<i>div</i>
<i>a</i>	<i>a</i>	<i>a</i>	<i>a</i>
<i>img</i>	<i>span</i>	<i>span</i>	<i>span</i>
<i>meta</i>	<i>li</i>	<i>li</i>	<i>li</i>
<i>br</i>	<i>img</i>	<i>img</i>	<i>img</i>
<i>table</i>	<i>script</i>	<i>script</i>	<i>script</i>
<i>td</i>	<i>p</i>	<i>p</i>	<i>p</i>
<i>tr</i>	<i>option</i>	<i>link</i>	<i>link</i>
	<i>i</i>	<i>meta</i>	
	<i>option</i>	<i>i</i>	
		<i>ul</i>	
		<i>option</i>	

Figure 3.10. Evolution of the most frequently used elements per page.

The top six elements haven't changed in the past three years, and it looks like the `link` element is gaining a foothold as a solid number seven.

It's interesting to see that `i` and `option` have both fallen out of favor. The first probably because libraries that misuse the `i` element for icons have fallen out of popularity in favor of libraries using SVGs for icons. The `meta` element is making a strong push into the top 10 this year, perhaps because social markup is also on the rise. We'll look at social markup in a later section of this chapter. The rise of styled `select` elements accounts for the `ul` (unordered list) element gaining popularity over the `option` element.

## main

With the creation of content spiking in 2021<sup>85</sup> (most likely because the world was stuck in a pandemic), we wanted to see if that correlates to an adoption of content elements as well. We

85. <https://wordpress.com/activity/posting/>

thought `main` is a good indicator, it being an informative element that doesn't affect the DOM's concept of the structure of a page.

# 27.9%

*Figure 3.11. Percent of mobile pages with at least one `main` element.*

27.7% of desktop pages and 27.9% of mobile pages had a `main` element. In terms of popularity, it made it well in the top 50 elements, at a respectable 34th place. Before you start thinking that there are only 114 elements, we've actually had more than a thousand elements come back from the queries we ran, most of which were custom.

## base

Another curiosity was how much developers were paying attention to the stricter rules of the HTML spec. For example, the spec says there must be no more than one `base` element in a document, because the `base` element defines how user agents should resolve relative URLs. Having more than one `base` element introduces ambiguity, so the spec requires that all `base` elements after the first be ignored, rendering them useless.

From looking at the desktop pages, `base` is a popular element, with 10.4% of pages having one. But do they have only one? There are 5,908 more `base` elements than pages, so we can only conclude at least some pages have more than one `base` element. Who said developers were great at following directions? We would also recommend people validate their HTML using the W3C-provided Markup Validation Service<sup>86</sup>.

## dialog

Throughout the chapter we wanted to also look at the adoption of some of the more controversial or new elements. `dialog` is one of them, with not all major browsers supporting it out of the box yet. Only 7,617 pages on desktop and 7,819 pages on mobile are using a `dialog` element. When we consider that's only around 0.1% of the pages analyzed, it doesn't look like the adoption is there yet.

86. <https://validator.w3.org/>

## canvas

The `canvas` element can be used with either the Canvas API<sup>87</sup> or WebGL API<sup>88</sup> to draw graphics and animations. It's one of the main elements used for games or mixed reality on the web. It's no surprise 3.1% of the desktop pages and 2.6% of the mobile pages use it. The higher usage on desktop makes sense when you consider the graphic capabilities of the different devices, and the use cases skewed towards games and virtual reality.

## Probability of element use

While the `html`, `head`, `body`, `title`, and `meta` elements are all optional, they're the most common elements this year, all present on more than 99% of the pages.

*Note that as we are looking at the rendered HTML, and the browsers will automatically add the `html` and `head` elements, this chart shows we have an error rate of 0.2% of pages in our crawl due to sites no longer being accessible at the time of the crawl.*

---

87. [https://developer.mozilla.org/docs/Web/API/Canvas\\_API](https://developer.mozilla.org/docs/Web/API/Canvas_API)  
88. [https://developer.mozilla.org/docs/Web/API/WebGL\\_API](https://developer.mozilla.org/docs/Web/API/WebGL_API)



Figure 3.12. Adoption of the top HTML elements.

While the percentages are slightly different when compared with last year, the order for the most popular elements remains the same. What about some of the more exotic elements?

<b>Element</b>	<b>Percent of pages (mobile)</b>
<code>tt</code>	0.04%
<code>ruby</code>	0.02%
<code>rt</code>	0.02%

Figure 3.13. Adoption of `tt`, `ruby`, and `rt` elements on mobile pages.

It's interesting to see that `tt`, a deprecated element for Teletype Text<sup>89</sup>, is 100% more popular than `ruby` and `rt`, which are the Ruby Annotation<sup>90</sup> and Text<sup>91</sup> elements still used for showing the pronunciation of East Asian characters.

## `script`

# 98.2%

Figure 3.14. Percent of mobile pages with at least one `script` element.

A little over 98% of the pages scanned contain at least one `script` element. It's no surprise that `script` is also the 6th most popular element on a page. Compared with last year, the `script` element seems to remain constant in terms of popularity and has slightly increased levels of occurrence in the millions of pages analyzed, from 97% to 98%.

# 51.4%

Figure 3.15. Percent of mobile pages with at least one `noscript` element.

51.4% of pages also contain a `noscript` element, which is generally used to display a message for browsers that have disabled JavaScript. Another popular use for the `noscript` element is the Google Tag Manager (GTM) snippet. 18.8% of pages on desktop and 16.9% of pages on mobile are using the `noscript` element as part of the GTM snippet. It's interesting to note that GTM is more popular on desktop than mobile.

89. <https://developer.mozilla.org/docs/Web/HTML/Element/tt>  
 90. <https://developer.mozilla.org/docs/Web/HTML/Element/ruby>  
 91. <https://developer.mozilla.org/docs/Web/HTML/Element/rt>

## template

One of the least recognized, but most powerful features<sup>92</sup> of the Web Components specification is the `template` element. Despite the fact that the `template` element is well supported on modern browsers since 2013, only 0.5% of the pages were using it in 2021. In terms of popularity, it didn't even make it into the top 50 elements. We thought this speaks volumes about the adoption curve of the modern HTML specification for web developers.

In case you don't really know what `template` does, here is a refresher from the specification: "the `template` element is used to declare fragments of HTML that can be cloned and inserted in the document by script". If you're a web developer and think that sounds familiar, you're right. Most of the popular frameworks today have a similar non-native mechanism to do the same: Angular has `ng-content`, React has portals<sup>93</sup> and Vue has `slot`. We would have thought those frameworks would use the native `template` element or Web Components instead of re-creating the functionality within the frameworks.

## style



Figure 3.16. Percent of mobile pages with at least one `style` element.

When creating a web page, three things come together. One is HTML, and we're looking at that throughout this chapter. The second one is JavaScript, and we saw in the previous section that the `script` element used to load JavaScript is one of the most popular ones. It doesn't come as a shock that the `style` element, used to inline CSS is similarly popular. 83.8% of the mobile pages scanned had at least one `style` element.

In terms of sheer popularity on a page, it barely made it into the top 20, with 0.7%. That leaves us to believe that while multiple `script` elements are popular on a page, most have five times fewer `style` elements on them. And that makes sense. Because `script` elements can be used for both inline and external scripts, but CSS uses a separate element, the `link` element, for loading external stylesheets. The `link` element is present on slightly more pages than the `script` element, while being slightly less popular in terms of the number of occurrences.

92. <https://css-tricks.com/crafting-reusable-html-templates/>

93. <https://reactjs.org/docs/portals.html>

## Custom elements

We've also looked at elements that didn't show up in the HTML or SVG spec, be it current or obsolete, to determine what custom elements were out there in the wild.

<b>Element</b>	<b>Number of pages</b>	<b>Percent of pages</b>
<code>rs-module-wrap</code>	123,189	2.0%
<code>wix-image</code>	76,138	1.2%
<code>pages-css</code>	75,539	1.2%
<code>router-outlet</code>	35,851	0.6%
<code>next-route-announcer</code>	9,002	0.1%
<code>app-header</code>	7,844	0.1%
<code>ng-component</code>	3,714	0.1%

Figure 3.17. Adoption of select custom elements on desktop pages.

By far, the most popular one is Slider Revolution<sup>94</sup>, with a majority of elements attributed to the framework. It more than tripled in popularity over the past year, which leads us to believe it might be a part of a popular template or site builder. A close second is Wix<sup>95</sup>, the popular free site builder. We initially couldn't identify `pages-css`, but Alon Kochba reached out and identified it as another custom element used by Wix, which also explains the similar page count to `wix-image`.

We would have thought that popular frameworks like Angular<sup>96</sup>, Next.js<sup>97</sup>, or the former Angular.js<sup>98</sup> would account for more custom components, but `router-outlet` and `ng-component` make up a small part of the custom component base.

## Obsolete elements

There are currently 28 obsolete and deprecated elements<sup>99</sup> described in the HTML reference. We wanted to see how many of those were still in use today. By far, the most used ones are `center` and `font`, and we're glad to see their usage has slightly declined when compared

94. <https://www.sliderrevolution.com/faq/developer-guide-output-class-tag-changes/>

95. <https://www.wix.com/>

96. <https://angular.io/>

97. <https://nextjs.org/>

98. <https://angularjs.org/>

99. [https://developer.mozilla.org/docs/Web/HTML/Element#obsolete\\_and\\_deprecated\\_elements](https://developer.mozilla.org/docs/Web/HTML/Element#obsolete_and_deprecated_elements)

with last year.

`nobr` and `big` on the other hand, while still being deprecated, have increased in usage slightly when compared with last year.



Figure 3.18. Adoption of the top obsolete HTML elements.

While the percentage of obsolete elements for mobile pages is slightly different when compared with desktop, the order remains the same.



Figure 3.19. Relative adoption of the top obsolete HTML elements.

Google still uses a `center` element on their homepage in 2021, but we're not going to judge.

## Proprietary and non-standard elements

While custom elements all have a hyphen in them, we've also encountered elements that are made up, don't have a hyphen, and don't show up on the HTML standard<sup>100</sup>.

100. <https://html.spec.whatwg.org/#toc-semantics>

<b>Element</b>	<b>Mobile</b>	<b>Desktop</b>
<i>jdiv</i>	0.8%	0.8%
<i>noindex</i>	0.9%	0.8%
<i>mediaelementwrapper</i>	0.6%	0.6%
<i>ymaps</i>	0.3%	0.2%
<i>h7</i>	0.1%	0.1%
<i>h8</i>	<0.1%	<0.1%
<i>h9</i>	<0.1%	<0.1%

Figure 3.20. Adoption of non-standard elements.

All of them were present last year as well, and can be attributed to popular frameworks or products like JivoChat, Yandex, MediaElement.js, and Yandex Maps. And because some people get carried away, or six is just not enough headers, `h7` to `h9`.

## Embedded content

<b>Element</b>	<b>Desktop</b>	<b>Mobile</b>
<i>iframe</i>	56.7%	54.5%
<i>source</i>	9.9%	8.4%
<i>picture</i>	6.1%	6.0%
<i>object</i>	1.4%	2.0%
<i>param</i>	0.4%	0.4%
<i>embed</i>	0.4%	0.4%

Figure 3.21. Adoption of elements for embedding content.

Content can be embedded through multiple elements in a page. The most popular is an `iframe`, followed at a considerable distance by `source` and `picture`.

The actual `embed` element is the least popular out of all the present elements for embedding

content.

## Forms

Forms, or ways of getting input from your visitors, are part of the fabric of the web. It's no surprise that 71.3% of pages on desktop and 67.5% of pages on mobile had at least one `form` on them. The most common occurrence was one (33.0% on desktop and 31.6% on mobile) or two (17.9% on desktop and 16.8% on mobile) `form` elements on a page.

A large, bold, blue number "4,256" with a comma separating the thousands. The digits are slightly shadowed, giving it a three-dimensional appearance.

Figure 3.22. The most `form` elements found on a single page.

There are also extreme cases with one page having 4,018 `form` elements on desktop and 4,256 `form` elements on mobile. We can't help but wonder what kind of input is so valuable, that you'd have to break it up in 4,000 pieces.

## Attributes

Element behaviors are heavily influenced by attributes, so we thought it was only fair we took a look at the attributes used on a page, explore `data-*` patterns, and some popular social attributes for `meta` elements.

## Top attributes



Figure 3.23. The most popular HTML attributes.

The most popular attribute is `class` and that's no surprise, given that it's used for styling. 34.3% of all the attributes found on the pages we queried were `class`. By contrast, `id` was much less used, at 5.2%. It's interesting to note that the `style` attribute edged out the `id` attribute in popularity, accounting for 5.6% of occurrences.

The second most popular attribute is `href`, with 9.9% of occurrences. With links being part of the fabric of the web, it's not surprising an anchor element attribute was this popular. What was surprising is that the `src` attribute was only twice as popular as the `alt` attribute, despite it being available to considerably more elements.<sup>101</sup>

<sup>101</sup> <https://developer.mozilla.org/docs/Web/HTML/Attributes>

## Meta flavors

`meta` elements are gaining some of their lost popularity this year, so we wanted to take a closer look at them. They provide a way to add machine-readable information to your pages, as well as perform some nifty HTTP equivalents. For example, setting a *Content Security Policy* for a page:

```
<meta http-equiv="Content-Security-Policy" content="default-src  
'self'; img-src https:///*;">
```

From the available attributes, `name` (paired with `content`) was the most popular. 14.2% of the `meta` elements did not have a `name` attribute. In conjunction with the `content` attribute, they are used as a key-value pair for passing in information. What information, you ask?

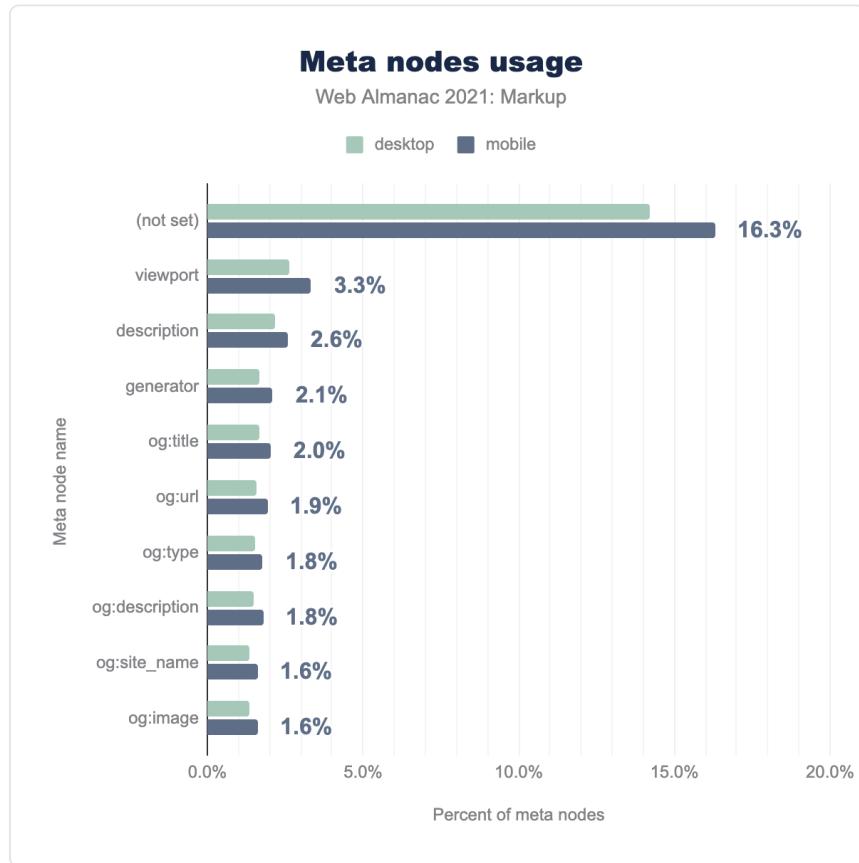


Figure 3.24. The most popular `meta` node names.

# 45.0%

Figure 3.25. Percent of `meta` viewports having a value of `initial-scale=1, width=device-width`.

The most popular is viewport information, with the most popular `viewport` value being `initial-scale=1, width=device-width`. 45.0% of mobile pages scanned used that value.

The second most popular combination are `og:*` meta elements, also known as Open Graph<sup>102</sup> meta elements. We'll talk about those in the next section.

<sup>102</sup>. <https://ogp.me/>

## Social markup

Providing information and assets for social platforms to use when previewing links to your page is a popular use case for the `meta` element.



Figure 3.26. Social `meta` nodes usage by page.

The most common by far are the Open Graph `meta` elements, used across multiple networks, with Twitter-specific elements lagging behind. `og:title`, `og:type`, `og:image`, and `og:url` are all required for every page, so it's interesting that there is a variation in their usage numbers.

## data- attributes

The HTML specification allows<sup>103</sup> for custom attributes, prefixed by `data-`. They are intended to store custom data, state, annotations, and the like, private to the page or application, for which there are no more appropriate attributes or elements.



Figure 3.27. The most popular `data-` attributes.

The most common ones, `data-id`, `data-src`, and `data-type` are non-specific, with `data-src`, `data-srcset`, and `data-sizes` being very popular with image lazy-loading libraries. `data-element_type` and `data-widget_type` are coming from a popular website builder, Elementor<sup>104</sup>.

103. [https://html.spec.whatwg.org/#embedding-custom-non-visible-data-with-the-data-\\*-attributes](https://html.spec.whatwg.org/#embedding-custom-non-visible-data-with-the-data-*-attributes)

104. <https://code.elementor.com/>

Slick, “the last carousel you’ll ever need”<sup>105</sup>, is responsible for `data-slick-index`. Popular frameworks like Bootstrap are responsible for `data-toggle`, while testing-library<sup>106</sup> is responsible for `data-testid`.

## Miscellaneous

We’ve covered a good chunk of the most common HTML use cases. We’ve set aside this section at the end to look into some of the more esoteric use cases, as well as adoption of new standards on the web.

### `viewport` specifications

The `viewport` `meta` element is used to control layout on mobile devices. Or at least that was the idea when it came out. Today, some browsers<sup>107</sup> have started to ignore some of the `viewport` options to allow for zooming a page up to 500%<sup>108</sup>.

---

105. <https://github.com/kenwheeler/slick>  
106. <https://testing-library.com/docs/queries/bytestid/>  
107. <https://www.quirksmode.org/blog/archives/2020/12/userscalableno.html>  
108. <https://dequeuniversity.com/rules/axe/4.0/meta-viewport-large>

Attribute	Desktop	Mobile
<code>initial-scale=1, width=device-width</code>	46.6%	45.0%
(empty)	12.8%	8.2%
<code>initial-scale=1, maximum-scale=1, width=device-width</code>	5.3%	5.6%
<code>initial-scale=1, maximum-scale=1, user-scalable=no, width=device-width</code>	4.6%	5.4%
<code>initial-scale=1, maximum-scale=1, user-scalable=0, width=device-width</code>	4.0%	4.3%
<code>initial-scale=1, shrink-to-fit=no, width=device-width</code>	3.9%	3.8%
<code>width=device-width</code>	3.3%	3.5%
<code>initial-scale=1, maximum-scale=1, minimum-scale=1, user-scalable=no, width=device-width</code>	1.9%	2.5%
<code>initial-scale=1, user-scalable=no, width=device-width</code>	1.89%	1.9%

Figure 3.28. Adoption of the most popular `meta` viewport values.

The most common `viewport` content option is `initial-scale=1, width=device-width`, which is not surprising when it's the recommended option on the MDN guide<sup>109</sup> explaining viewports. 45.0% of the pages analyzed are using it, almost 3% more than last year<sup>110</sup>. 8.2% of pages had an empty `content` attribute, slightly more than last year as well. That correlates with a decrease in usage for improper combinations of viewport options.

## Favicons

Favicons are one of the most resilient pieces of the web. They work even without markup and accept multiple image formats. There are also literally dozens of sizes you need to use to be thorough.

109. [https://developer.mozilla.org/docs/Web/HTML/Viewport\\_meta\\_tag](https://developer.mozilla.org/docs/Web/HTML/Viewport_meta_tag)

110. <https://almanac.httparchive.org/en/2020/markup#viewport-specifications>



*Figure 3.29. The most popular favicon formats.*

There were a few surprises when we looked at the data:

- ICO was finally dethroned as the most popular format by PNG.
- JPG is still used, even though it's not the best option when compared with some of the other unpopular options.
- With SVG support for favicons finally improving, SVG has overtaken WebP this year in terms of popularity.

## Button and input types

# 65.5%

Figure 3.30. Percent of mobile pages with at least one `button` element.

Buttons are controversial. There are a lot of opinions about what does and what doesn't constitute a button on the web. While we're not taking sides, we thought we should look at some of the semantic ways to specify a `button` element, seeing as how 65.5% of pages already had a `button` element on them.

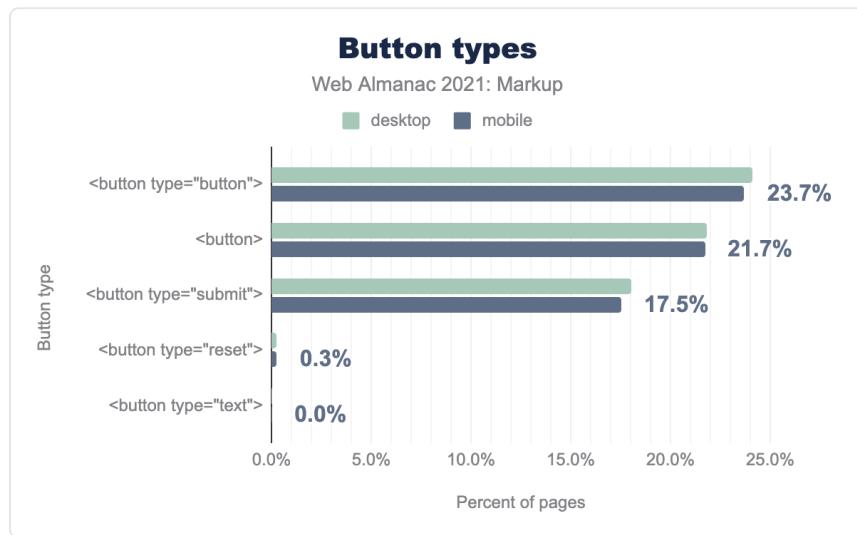


Figure 3.31. The most popular button types.

When we compared the data to last year<sup>111</sup>, we noticed a lot more pages had `button` elements on them. This year we didn't run a query for `input`-typed buttons, but we've seen a definite decrease in usage for the number of `button` elements on pages. The Accessibility chapter also has a whole section on buttons, you should read that as well!

111. <https://almanac.httparchive.org/en/2020/markup#button-and-input-types>

## Links

Link	Desktop	Mobile
Always uses <code>target="_blank"</code> with <code>noopener</code> and <code>noreferrer</code>	22.0%	23.2%
Sometimes uses <code>target="_blank"</code> with <code>noopener</code> and <code>noreferrer</code>	78.0%	76.8%
Has <code>target="_blank"</code>	81.2%	79.9%
Has <code>target="_blank"</code> with <code>noopener</code> and <code>noreferrer</code>	14.3%	13.2%
Has <code>target="_blank"</code> with <code>noopener</code>	21.2%	20.1%
Has <code>target="_blank"</code> with <code>noreferrer</code>	1.2%	1.1%
Has <code>target="_blank"</code> without <code>noopener</code> and <code>noreferrer</code>	71.1%	69.9%

Figure 3.32. Adoption of various combinations of link attributes.

Links are the glue that ties the web together. Normally, we wanted to look at the instances where they are proving problematic. Using `target="_blank"` without `noopener` and `noreferrer` was a security vulnerability for the longest time, but 71.1% of desktop pages and 68.9% of mobile pages still use it today.

That's what probably prompted a spec change<sup>12</sup> this year, so now browsers set `rel="noopener"` by default on all `target="_blank"` links.

## Web Monetization

Web Monetization<sup>13</sup> is being proposed as a W3C standard at the Web Platform Incubator Community Group<sup>14</sup> (WICG). It's a young standard that provides an open, native, efficient, and automatic way to compensate creators, pay for API calls, and support crucial web infrastructure. While it is in its early days, and it is not implemented by any of the major browsers, it is supported via forks and extensions, and has been instrumented in Chromium and the HTTP Archive dataset for over a year. We wanted to take a look at adoption so far.

12. <https://github.com/whatwg/html/issues/4078>

13. <https://discourse.wicg.io/t/proposal-web-monetization-a-new-revenue-model-for-the-web/3785>

14. <https://www.w3.org/community/wicg/>

# 1,067

Figure 3.33. Number of mobile pages that use Web Monetization.

Web Monetization popularly uses a `meta` element on the page, specifying the wallet address for the money to be paid into. It looks a little bit like:

```
<meta name="monetization" content="$wallet.example.com/alice">
```



Figure 3.34. Adoption of Web Monetization over time. (Source: Chrome Status<sup>115</sup>)

While it still seems a vanishingly small number by percentages, it has shown growth—more on desktop than mobile. It's important to keep in mind how big the HTTP Archive dataset is and how slowly it takes to gain numbers, even for a feature that is widely and natively supported. It will be interesting to continue to track these numbers and developments over more time. This author might be biased, as an editor for the Web Monetization standard, but you're encouraged to give it a try<sup>116</sup>, it's free.

There has been an issue open for some time<sup>117</sup>, and the new version of the specification will use a `link` instead. Only 36 pages in our desktop set and 37 in our mobile set used the `Link` version, and all of those also included the `meta` version as well.

We know there are currently two Interledger<sup>118</sup>-enabled wallet providers in the ecosystem, so

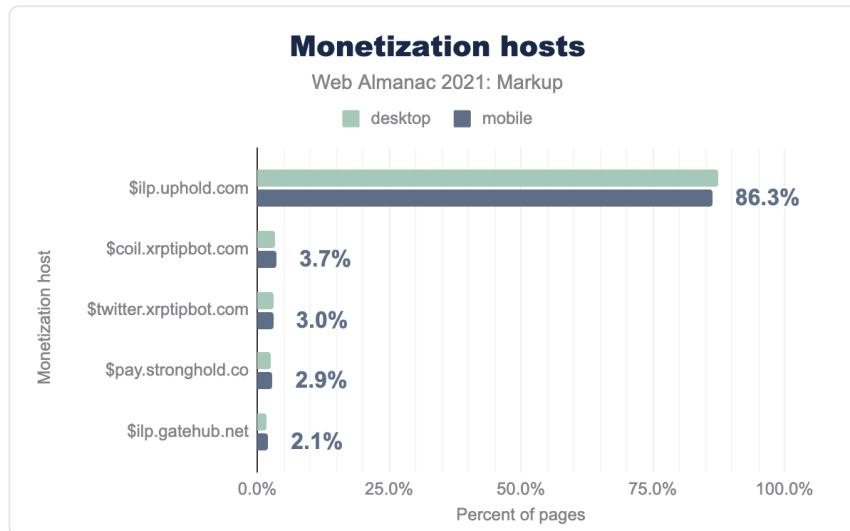
115. <https://www.chromestatus.com/metrics/feature/timeline/popularity/3119>

116. <https://webmonetization.org/docs/getting-started>

117. <https://github.com/WICG/webmonetization/issues/19>

118. <https://interledger.org/>

we wanted to see the distribution and adoption of those wallets.



*Figure 3.35. The most popular Web Monetization hosts.*

Uphold and Gatehub are the current wallets, and it looks like Uphold is the dominant wallet by far. What is curious, a wallet that was deprecated this year, Stronghold, was more popular than an active wallet provider, Gatehub. We thought that speaks towards the rate at which web developers update their web sites.

## Conclusion

We've pointed out interesting, surprising, and concerning bits of data throughout the chapter. Let us reflect once more on the state of markup in 2021.

The most surprising for us was that, almost 20 years later, XHTML was still used on a considerable part of the web, with a little over 7% of pages using it in 2021.

The median page sizes in 2020 were shrinking when compared to 2019, but this year it looks like the trend has regressed, surpassing the median sizes for 2019 as well. The web is getting heavier. Again.

English is relatively less popular on mobile pages. We're not sure why, and this author would like to encourage you to explore the possibilities of why this is the case.

It was interesting to see that libraries adopting better practices correlated directly with

elements falling out of favor. Both `i` and `option` are less-used this year because icon libraries have switched over to using SVG.

It was great to see ICO finally being dethroned as the most popular favicon format in favor of PNG. Similarly, seeing SVG more than doubling in usage for favicons in the past year made us think we're 10 years away from dethroning PNG.

The `doctype` percentage has increased steadily by half a percentage point every year. At this rate, we'll live in an ideal world where every page has a `doctype` by 2027.

It was concerning for this author to see that the adoption of some of the newer standards is slow, sometimes on a 10-year cycle, and that web pages don't get updated as often as we'd like.

*With that in mind, I'll leave you to reflect on the state of the web in 2021. I'd also encourage you to be part of the people who increase adoption of new standards every year. Start with something new you've learned today, one of the many standards we've covered not only in this chapter but in this whole Web Almanac publication.*

## Author



### Alex Lakatos

🐦 @avolakatos    🌐 AlexLakatos    ⚡ http://alexlakatos.com/

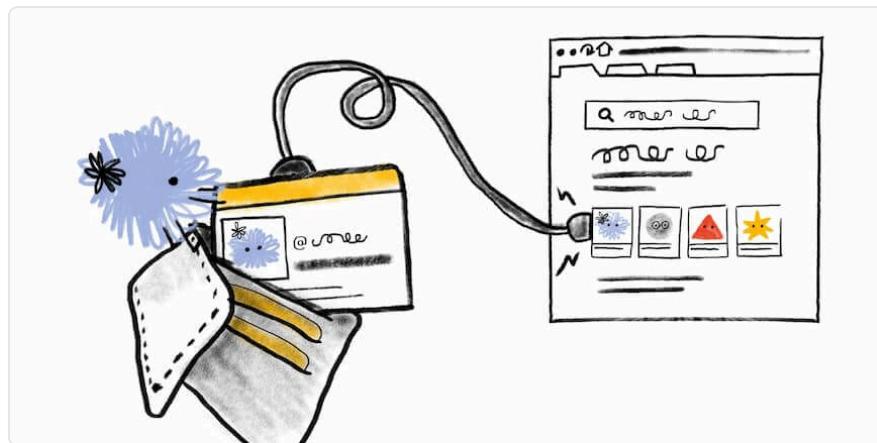
Alex Lakatos has spent the past decade working on the Open Web within Browser, Communications, and FinTech organizations. With a background in web technologies and developer advocacy, he's helping the Interledger Foundation<sup>119</sup> build developer-friendly products while engaging with the developer community at large. You can reach out to him on Twitter<sup>120</sup>.

119. <https://interledger.org/>  
120. <https://twitter.com/avolakatos>



# Part I Chapter 4

# Structured Data



*Written by Jono Alderson and Andrea Volpini*

*Reviewed by Koen Van den Wijngaert and Phil Barker*

*Analyzed by Greg Brimble*

*Edited by Jarno van Driel, Jasmine Drudge-Willson, and Barry Pollard*

## Introduction

When reading web pages, we consume *unstructured* content. We read paragraphs, examine media, and consider what we digest. As part of that process, we apply intuition and context (such as subject-matter familiarity) to identify key themes, data points, entities, and relationships. As humans, we're very good at this.

But this kind of intuition and context is difficult for software to replicate. It's difficult for systems to reliably parse, identify, and extract key themes with a high degree of reliability.

These limitations can constrain the kinds of things which we can effectively build and create, and limits how "smart" web technology can be.

By introducing *structure* to information, we can make it *much* easier for software to understand content. We do this by adding labels and metadata which identify key concepts and entities—as well as their properties and relationships.

When machines can reliably extract structured data, at scale, we enable new and smarter types of software, systems, services and businesses.

The goal of the Web Almanac's Structured Data chapter is to explore how structured data is currently being used across the web. We hope that this will provide insight into the landscape, the challenges, and the opportunities at hand.

This is the first time that this chapter has been included in the Web Almanac, and so we unfortunately lack historical data for the purposes of comparison. Future chapters will also explore year-on-year trends.

## Key concepts

Structured data is a complex landscape, and one which is by nature abstract and 'meta'. To understand the significance and potential impact of structured data, it's worth exploring the following key concepts.

### The semantic web

When we add structured data to public web pages—and we define the entities that those pages contain (or are about, or reference)—we create a form of linked data<sup>121</sup>.

We make *statements* about the things in (and related to) our content in the form of *triples*. Statements like, "This **article** was **authored** by this **person**", or "That **video** is **about** a **cat**".

Describing our content in this way enables machines to treat web pages and websites as databases. At scale, it creates a semantic web<sup>122</sup>; a giant global database of information.

The Semantic Web is the name of a long-term project started by W3C with the stated purpose of realizing the idea of having data on the Web defined and linked in a way that it can be used by machines not just for display purposes, but for automation, integration, and reuse of data across various applications

— Greg Ross, An introduction to Tim Berners-Lee's Semantic Web<sup>123</sup>

That creates a wealth of possibilities for business, technology, and society.

---

121. [https://en.wikipedia.org/wiki/Linked\\_data](https://en.wikipedia.org/wiki/Linked_data)

122. <https://www.techrepublic.com/article/an-introduction-to-tim-berners-lees-semantic-web/>

## Search engines, and beyond

To date, some of the broadest consumers of structured data are *search engines* and *social media platforms*.

In most major search engines, website owners may become eligible for various forms of *rich results* (which may influence visibility and traffic) by implementing various types of structured data on their websites.

In fact, search engines have played such a significant role in the general adoption of (and education around<sup>124</sup>) structured data across the web, that this chapter was born out of Web Almanac SEO chapters from previous years<sup>125</sup>. In recent years, the influence of search engines has also popularized schema.org<sup>126</sup> the vocabulary of choice for structured data.

In addition to this, social media platforms rely on structured data to influence how they read and display content when it's shared (or linked to) on their platforms. Rich previews, tailored titles and descriptions, and interactivity in these platforms are often powered by structured data.

But there's more to see and understand here than search engine optimization and social media benefits. The scale, variety, impact and *potential* of structured data goes far beyond rich results, far beyond search engines, and far beyond schema.org.

For example, structured data facilitates:

- Easier topic modelling and clustering across multiple pages, websites and concepts; enabling new types of research, comparison and services.
- Enriching analytics data, to allow for deeper and horizontalized analysis of content and performance.
- Creating a unified (or at least, connected) language and syntax for querying business systems and website content.
- Semantic search; using the same rich metadata used for search engine optimization, to create and manage internal search systems.

Whilst the findings of our research are inevitably shaped by the influence of search engines, we hope to also explore other types, formats, and use-cases of structured data.

---

124. <https://developers.google.com/search/docs/advanced/structured-data/intro-structured-data>

125. <https://almanac.httparchive.org/en/2020/seo#structured-data>

126. <https://schema.org/>

## Types of structured data and coverage

Structured data comes in many formats, standards, and syntaxes. We've collected data about the most common of these across our data set.

Specifically, we've identified and extracted structured data relating to:

- Schema.org<sup>127</sup>
- Dublin core<sup>128</sup>
- Meta tags used by social networks:
  - Open Graph<sup>129</sup>
  - Twitter<sup>130</sup>
  - Facebook<sup>131</sup>
- Microformats<sup>132</sup> (and microformats2<sup>133</sup>)
- RDFa<sup>134</sup>, Microdata<sup>135</sup> and JSON-LD<sup>136</sup>

Collectively, these provide a broad overview of different use-cases and scenarios; and include both legacy standards and modern approaches (e.g., microformats vs JSON-LD).

Before we explore specific usage across the various structured data types, we should briefly explore some caveats.

### Data caveats

#### 1. The influence of Content Management Systems

Many of the pages we've evaluated are from websites which use a Content Management System (CMS), such as WordPress<sup>137</sup> or Drupal<sup>138</sup>. These systems—or the themes/plugins/modules which enhance their functionality—are often responsible for generating the HTML markup which contains the structured data which we're analyzing.

---

127. <http://schema.org/>  
 128. <https://www.dublincore.org/specifications/dublin-core/>  
 129. <https://ogp.me/>  
 130. <https://developer.twitter.com/en/docs/twitter-for-websites/cards/guides/getting-started>  
 131. <https://developers.facebook.com/docs/sharing/webmasters/>  
 132. <http://microformats.org/>  
 133. <https://microformats.org/wiki/microformats2>  
 134. <https://en.wikipedia.org/wiki/RDFa>  
 135. [https://en.wikipedia.org/wiki/Microdata\\_\(HTML\)](https://en.wikipedia.org/wiki/Microdata_(HTML))  
 136. <https://json-ld.org/>  
 137. <https://wordpress.org/>  
 138. <https://www.drupal.org/>

That means that our findings are unavoidably skewed to aligning with the behaviors and output of the most prevalent CMS'. For example, many websites using Drupal automatically output structured data in the form of RDFa, and WordPress (which powers a significant percentage of websites) often includes microformats markup in template code. This contributes significantly to the shape of our findings.

## 2. The limitations of homepage-only data

Unfortunately, the nature and scale of our data-collection methods limit our analysis to homepages only (i.e., the root URL of each hostname we evaluate).

This significantly limits the amount of data we can collect and analyze, and undoubtedly skews the kinds of data we've collected.

As most homepages act as portals to more specific pages, we can reasonably expect that our analysis underestimates the prevalence of the kinds of content present on that deeper pages. That likely includes information relating to *articles*, *people*, *products* and similar.

Conversely, we likely over-index on information typically found on homepages, and site-wide information which is present on all pages—like information about *web pages*, *websites* and *organizations*.

## 3. Data overlaps

The nature of some structured data formats makes it hard to perform this kind of analysis cleanly at scale. In many cases, structured data is implemented in multiple (often overlapping) formats, and the lines between syntaxes and vocabularies get blurred.

For example, Facebook and Open Graph metadata are technically a subset of RDFa. That means that our research identifies a page containing a Facebook meta tag in our Facebook category, and our [RDFa](#) section. We've done our best to clean, normalize, and make sense of these types of overlaps and nuances.

## 4. Mobile metrics

Throughout our data set, the adoption and presence of structured data varies only very slightly between our desktop and mobile data sets. As such, for the sake of brevity, our narrative focuses predominantly on the *mobile* data set.

## Usage by type

We can see that there's a broad range of different types of structured data across many of the pages in our set.



Figure 4.1. Structured data usage

We can also see that *RDFa* and *Open Graph* tags in particular are extremely prevalent, appearing on 60.61% and 57.45% of pages respectively.

At the other end of the scale, legacy formats, like *Microformats* and *microformats2*, appear on fewer than 1% of pages.

## Coverage by syntax type

In addition to identifying when a certain type of structured data is present, we collect information on the types of data it describes. We can break each of these down and explore how each format and syntax is being used.

### RDFa

Resource Description Framework in Attributes<sup>139</sup> (*RDFa*) is a technology for linked data markup, which was introduced by W3C in 2015. It allows users to augment and translate visual

139. <https://www.w3.org/TR/rdfa-lite/>

information on a web page by adding additional attributes to markup.

For example, a website owner might add a `rel="license"` attribute to a hyperlink in order to explicitly describe it as a link to a licensing information page.



Figure 4.2. RDFa types

When we evaluate the types of RDFa, we can see that the `foaf:image` syntax is present on far more pages than any other type—on upwards of 0.86% of all pages in our data set. Whilst that may seem like a small proportion, it represents over ~65,000 pages, and over 60% of the total RDFa markup that we discovered.

Beyond this outlier, the use of RDFa diminishes and fragments considerably, though there are still some interesting discoveries to explore.

## On FOAF

FOAF<sup>140</sup> (or “Friend of a Friend”) is a linked data dictionary of people-related terms, created in

140. <http://xmlns.com/foaf/spec/>

the early-2000s. It can be used to describing *people, groups and documents*.

FOAF uses W3C's RDF syntax and in its original introduction<sup>141</sup> was explained as follows:

*Consider a Web of inter-related home pages, each describing things of interest to a group of friends. Each new home page that appears on the Web tells the world something new, providing factoids and gossip that make the Web a mine of disconnected snippets of information. FOAF provides a way to make sense of all this.*

#### *Introducing FOAF<sup>142</sup>*

Anecdotally, we can attribute a prominence of `foaf` markup in our results to sites running on older versions of the Drupal CMS, which historically added `typeof="foaf:image"` and `foaf:document` markup to its HTML by default.

### **On other notable RDFa findings**

As well as FOAF properties, various other standards and syntaxes show up in our list.

Notably, we can see several `sioc` properties, such as `sioc:item` (0.24% of pages) and `sioc:useraccount` (0.03% of pages). SIOC<sup>143</sup> is a standard designed to describe structured data relating to *online communities*, such as message boards, forums, wikis and blogs.

We can also see a SKOS<sup>144</sup> (or "Simple Knowledge Organization System") property—`skos:concept`—on 0.04% of pages. SKOS is another standard, which aims to provide a way of describing taxonomies and classifications (e.g., tags, data sets, and so on).

### **Dublin Core**

Dublin Core<sup>145</sup> is a vocabulary interoperable with linked data standards that was originally conceived in Dublin, Ohio in 1995 at an OCLC (Online Computer Library Center) and NCSA (National Center for Supercomputing Applications) workshop.

It was designed to describe a broad range of resources (both digital and physical) and can be used in various business scenarios. Starting in 2000 it became extremely popular among RDF-based vocabularies and received the adoption of the W3C.

141. <https://web.archive.org/web/20140331104046/http://www.foaf-project.org/original-intro>

142. <https://web.archive.org/web/20140331104046/http://www.foaf-project.org/original-intro>

143. <https://www.w3.org/Submission/sioc-spec/>

144. <https://www.w3.org/TR/skos-primer/>

145. <https://dublincore.org/>

Since 2008 it is managed by the Dublin Core Metadata Initiative (DCMI) and remains highly interoperable with other linked data vocabularies. It is typically implemented as a collection of meta tags in an HTML document.



Figure 4.3. Dublin Core usage

That the most popular attribute type is `dc:title` (on 0.70% of pages) comes as no surprise; but it is interesting to see that `dc:language` is next (above common descriptors like `description`, `subject` and `publisher`) with a penetration of 0.49%. This makes sense, when you consider that Dublin Core is often used in multilingual metadata management systems.

It's also interesting to see the relatively prominent appearance of `dc:relation` (on 0.16% of pages)—an attribute that is capable of expressing relationships between different concepts.

While it might seem to many that Schema.org is predominant in the context of SEO, the role of DC remains pivotal because of its broad interpretation of concepts and its deep roots in the *linked open data movement*.

## Social metadata

Social networks and platforms are some of the biggest publishers and consumers of structured data. This section explores the roles, breadth of adoption, and scale of some of their specific structured data formats.

### Open Graph

The Open Graph protocol<sup>146</sup> is an open-source standard, originally created by Facebook. It is a type of structured data specific to the context of *sharing content*, based loosely on Dublin Core, Microformats and similar standards.

It describes a series of meta tags and properties, which may be used to define how content should be (re)presented when shared between platforms. For example, when liking or embedding a post, or sharing a link.

These tags are typically implemented in the `<head>` of an HTML document, and define elements such as the page's *title*, *description*, *URL*, and *featured image*.

The Open Graph protocol has since been broadly adopted by many platforms and services, including *Twitter*, *Skype*, *LinkedIn*, *Pinterest*, *Outlook* and more. When platforms don't have their own standards for how shared/embedded content should be presented (and sometimes, even when they do), Open Graph tags are often used to define the default behavior.

---

146. <https://ogp.me/>

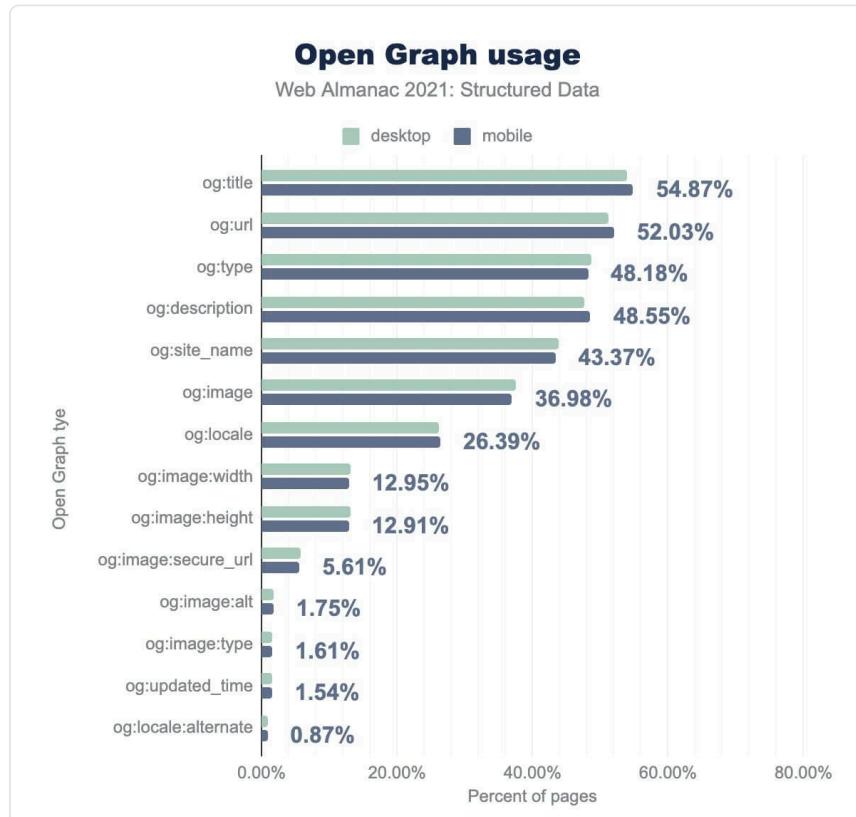


Figure 4.4. Open Graph usage

The most common type of Open Graph tag is the `og:title`, which can be found on an incredible 54.87% of pages. That's followed closely by a set of related attributes, which describe what type of thing is being represented (e.g., `og:type`, on 48.18% of pages) and how it should be represented (e.g., `og:description`, on 48.55% of pages).

This narrow distribution is to be expected, as these tags are often used together as part of a "boilerplate" set of tags used in the `<head>` across all pages on a site.

Slightly less common is `og:locale` (26.39% of pages), which is used to define the language of the page's content.

Less common still is more specific metadata about the `og:image` tag, in the form of `og:image:width` (12.95% of pages), `og:image:height` (12.91% of pages), `og:image:secure_url` (5.61% of pages) and `og:image:alt` (1.75% of pages). It's worth noting that with HTTPS adoption now increasingly the norm, `og:image:secure_url` (which

was intended to identify a `https://` version of the `og:image`) is now largely redundant.

Beyond these examples, usage drops off rapidly, into a long tail of (often malformed, deprecated or erroneous) tags.

## Twitter

Though Twitter uses Open Graph tags as fallbacks and defaults, the platform supports its own flavor of structured data. A set of specific meta tags (all prefixed with `twitter:`) can be used to define how pages should be presented when URLs are shared on Twitter.



Figure 4.5. Twitter meta tag usage

The most common Twitter meta tag is `twitter:card`, which was found on 35.42% of all pages. This tag can be used to define how pages should be presented when shared on the platform (e.g., as a *summary*, or as a *player* when paired with additional data about a media

object).

Beyond this outlier, adoption drops off steeply. The next most common tags are `twitter:title` and `twitter:description` (both also used to define how shared URLs are presented), which appear on 20.86% and 18.68% of all pages, respectively.

It's understandable why these particular tags—as well as the `twitter:image` tag (11.41% of pages) and `twitter:url` tag (3.13% of pages)—aren't more prevalent, as Twitter falls back to the equivalent Open Graph tags (`og:title`, `og:description` and `og:image`) when they're not defined.

Also of interest are:

- The `twitter:site` tag (11.31% of pages) which defines the Twitter account associated with the website in question.
- The `twitter:creator` tag (3.58% of pages), which defines the Twitter account of the author of the web page's content.
- The `twitter:label1` and `twitter:data1` tags (both on 6.85% of pages), which can be used to define custom data and attributes about the web page. Additional label/data pairs (e.g., `twitter:label2` and `twitter:data2`) are also present on a significant number (0.5%) of pages.

Beyond these examples, usage drops off rapidly, into a long tail of (often malformed, deprecated or erroneous) tags.

## Facebook

In addition to Open Graph tags, Facebook supports additional metadata (meta tags, prefixed with `fb:`) for relating web pages to specific brands, properties and people on their platform.



Figure 4.6. Facebook meta tag usage

Of all of the Facebook tags that we detected, there are only three tags with significant adoption.

Those are `fb:app_id`, `fb:admins`, and `fb:pages`; which we found on 6.06%, 2.63% and 0.86% of pages respectively.

These tags are used to explicitly relate a web page to a Facebook Page/Brand, or to grant permissions to a user (or users) who administrates those profiles.

Anecdotally, it's unclear how well these are supported by Facebook. The platform has gone through radical changes over the past few years, and their technical documentation hasn't been well-maintained. However, many content management systems, templates and best practice guides—as well as some of Facebook's debugging tools—still include and make reference to them.

## Microformats and microformats2

Microformats (commonly abbreviated as `μF`) are an open data standard for metadata to embed semantics and structured data in HTML.

They are composed of a set of defined classes that describe the meanings behind normal HTML elements, such as headings and paragraphs.

The guiding principle behind this format for structured data is to convey semantics by reusing widely adopted standards (semantic (X)HTML). The official documentation<sup>147</sup> describes Microformats as “designed for humans first and machines second”, and are “a set of simple, open data formats built upon existing and widely adopted standards”.

Microformats are available in two versions: Microformats v1 and Microformats v2 (microformats2). The latter, introduced in March 2014, replaces and supersedes v1 and takes advantage of some important lessons learned from both microdata and RDFa syntaxes.



Figure 4.7. Microformats usage

Historically and due to its nature (as an extension of HTML), Microformats have been heavily used by website developers to describe properties of businesses and organizations; particularly in pages promoting local businesses. This goes a long way to explaining the prominence of the `adr` property (on 0.50% of pages), reviews (`hReview`, on 0.06% of pages) and other information meant to characterize local businesses and their products/services.

147. <https://microformats.org/wiki/what-are-microformats>

Figure 4.8. *microformats2 usage*

The difference between legacy microformats and the more modern version is significant, and an interesting insight into changing behaviors and preferences in the use of markup.

Where the `adr` class dominated the classic microformats data set, the equivalent `h-adr` property only occurs on 0.02% of pages. The results here are dominated instead by the `h-entry` property (on 0.08% of pages and which describes blog posts and similar content units), and the `h-card` property (on 0.04% of pages and which describes a *business card* of an organization or individual).

We can speculate on three likely causes for this difference:

- Data for common class names (like `adr`) is almost certainly over-inflated in our microformats v1 data; where it's difficult to distinguish between when these values are used for *structured data* vs more *generic* reasons (e.g., as an HTML class attribute value with associated CSS rules).
- The use of microformats in general (regardless of type) has decreased significantly, and been replaced with other formats.
- Many websites and themes still include `h-entry` (and sometimes `h-card`) markup on common design elements and layouts. For example, many WordPress themes continue to output a `h-entry` class on the main content container.

## Microdata

Like microformats and RDFa, microdata<sup>148</sup> is based on adding attributes to HTML elements. Unlike microformats, but in common with RDFa, it's not tied to a set of defined meanings. The standard is extensible and allows authors to declare which vocabularies of data they're describing; most commonly schema.org.

One of the limitations of microdata is that it can be difficult to describe abstract or complex relationships between entities, when those relationships aren't explicitly reflected in the HTML structure of the page.

For example, it may be hard to describe the *opening hours* of an *organization* if that information isn't concurrent or logically structured in the document. Note that, there are standards and methodologies for solving this problem (e.g., by including inline `<meta>` tags and properties), but these aren't widely adopted.

---

148. [https://en.wikipedia.org/wiki/Microdata\\_\(HTML\)](https://en.wikipedia.org/wiki/Microdata_(HTML))



Figure 4.9. Microdata types

The most common types of microdata across the pages we analyzed describe the web page itself; via properties like `webpage` (7.44% of pages), `sitenavigationelement` (5.62% of pages), `wpheader` (4.87% of pages) and `wpfooter` (4.56% of pages).

It's easy to speculate on why these types of structural descriptors are more prominent than content descriptors (such as `person` or `product`); creating and maintaining microdata requires content producers to add specific code to their content—and that's often easier to do at template level than it is at content level.

Whilst one of the strengths of microdata is its explicit relationship with (and authoring in) the HTML markup, this has limited its approach to content authors with the technical knowledge and capabilities to use it.

That said, we see a broad adoption and variety of microdata types. Of note:

- `Organization` (4.02%), which typically describes the company which *publishes* the website, the *manufacturer* of a product, the *employer* of an author, or similar.
- `CreativeWork` (2.14%) the most generic parent type to describe all written and visual content (e.g., blog posts, images, video, music, art).
- `BlogPosting` (1.34%), which describes an individual blog post (which commonly also identifies a `Person` as an author).
- `Person` (1.37%) which is often used to describe content authors and people related to the page (e.g., the publisher of the website, the owner of the publishing organization, the individual selling a product, etc.).
- `Product` (1.22%) and `Offer` (1.09%), which, when used together, describe a product which is available for purchase (typically with additional properties which describe pricing, reviews and availability).

## JSON-LD

Unlike microdata and microformats, JSON-LD<sup>149</sup> isn't implemented by adding properties or classes to HTML markup. Instead, machine-readable code is added to the page as one or more standalone blobs of JavaScript Object Notation. This code contains descriptions of the entities on the page, and their relationships.

Because the implementation isn't tied directly to the HTML structure of the page, it can be much easier to describe complex or abstract relationships, as well as representing information which isn't readily available in the human-readable content of the page.

149. <https://json-ld.org/>

*Figure 4.10. JSON-LD usage*

As we might expect, our findings are similar to our findings from evaluating the use of microdata. That's to be expanded, as both approaches are heavily skewed towards the use of schema.org as a predominant standard. However, there are some interesting differences.

Because the JSON-LD format allows for site owners to describe their content independently of the HTML markup, it can be easier to represent more abstract complex relationships, which aren't tied so strictly to the content of the page.

We can see this reflected in our findings, where more specific and structured descriptors are more common than with microdata. For example:

- `BreadcrumbList` (1.45% of pages) describes the hierarchical position of the web page on the website (and describes each parent page).
- `ItemList` (0.5% of pages), which describes a set of entities, such as *steps* in a *recipe*, or *products* in a *category*.

Outside of these examples, we continue to see a similar pattern as we did with microdata (though at a much lower scale). Descriptions of websites, local businesses, organizations and the structure of web pages account for the majority of broad adoption.

### **JSON-LD structures & relationships**

One key advantage of JSON-LD is that we can more easily describe the relationships between entities than we can in other formats.

An event, for example, may have an organizing *corporation*, be located at a specific *location*, and have tickets available on sale as part of an *offer*. A *blog post* describing that event might have an *author*, and so on, and so on. Describing these kinds of relationships is much easier with JSON-LD than with other syntaxes and can help us tell rich stories about entities.

However, these relationships can often become deep, complex and intertwined. So, for the purposes of this analysis, we're only looking at the most common types of relationships between entities; not evaluating entire trees and relationship structures.

Below are the most common connections between types, based on how frequently they occur within all structure/relationship values. Note that some of these structures and values may sometimes overlap, as they're small parts of larger relationship chains.

<b>Relationship</b>	<b>% of desktop pages</b>	<b>% of mobile pages</b>
<i>WebSite &gt; potentialAction &gt; SearchAction</i>	6.44%	6.15%
	5.06%	4.85%
<i>@graph &gt; WebSite</i>	4.89%	4.69%
<i>WebPage &gt; isPartOf &gt; WebSite</i>	4.02%	3.81%
<i>@graph &gt; WebPage</i>	4.01%	3.79%
<i>BreadcrumbList &gt; itemListElement &gt; ListItem</i>	3.93%	3.78%
<i>Organization &gt; logo &gt; ImageObject</i>	2.85%	3.03%
<i>@graph &gt; BreadcrumbList</i>	3.18%	2.99%
<i>WebPage &gt; potentialAction &gt; ReadAction</i>	2.92%	2.71%
<i>WebPage &gt; breadcrumb &gt; BreadcrumbList</i>	2.60%	2.44%
<i>WebSite</i>	2.49%	2.30%
<i>@graph &gt; Organization</i>	2.26%	2.13%
<i>WebSite &gt; publisher &gt; Organization</i>	2.22%	2.09%
<i>Product &gt; offers &gt; Offer</i>	1.47%	1.89%
<i>Product</i>	1.41%	1.73%
<i>@graph &gt; ImageObject</i>	1.80%	1.71%
<i>ItemList &gt; itemListElement &gt; ListItem</i>	1.71%	1.69%
<i>@graph &gt; SiteNavigationElement</i>	1.70%	1.66%
<i>WebPage &gt; primaryImageOfPage &gt; ImageObject</i>	1.67%	1.59%

Figure 4.11. JSON-LD entity relations.

The most common structure is the relationship between `website`, `potentialAction`, and `SearchAction` schema (accounting for 6.15% of structures). Collectively, this relationship enables the use of a *Sitelinks Search Box* in Google's search results.

Perhaps most interestingly, the next most popular structure (4.85% of relationships) defines no relationships. These pages output only the simplest types of structured data, defining individual, isolated entities and their properties.

The next most popular structure (4.69% of relationships) introduces the `@graph` property (in conjunction with describing a `website`). The `@graph` property doesn't is not an entity in its own right but can be used in JSON-LD to contain and group relationships between entities.

As we explore further relationships, we can see various descriptions of content and organizational relationships, such as `WebPage > isPartOf > WebSite` (3.81% of relationships), `Organization > logo > ImageObject` (3.03% of relationships), and `WebSite > publisher > Organization` (2.09% of relationships).

We can also see lots of structures related to breadcrumb navigation, such as:

- `BreadcrumbList > itemListElement > ListItem` (3.78% of relationships)
- `@graph > BreadcrumbList` (2.99% of relationships)
- `ItemList > itemListElement > ListItem` (1.69% of relationships)

Beyond these most popular structures, we see an extremely long-tail of relationships, describing all manner of entities, content types and concepts; as niche as `ApartmentComplex > amenityFeature > LocationFeatureSpecification` (0.1% of relationships) and `AutoDealer > department > AutoRepair` (0.04% of relationships) and `MusicEvent > performer > PerformingGroup` (0.01% of relationships).

We should reiterate that these types of structures and relationships are likely to be much more common than our data set represents, as we're limited to analyzing the homepages of websites. That means that, for example, a website which lists many thousands of individual apartment complexes, but does so on inner pages, wouldn't be reflected in this data.

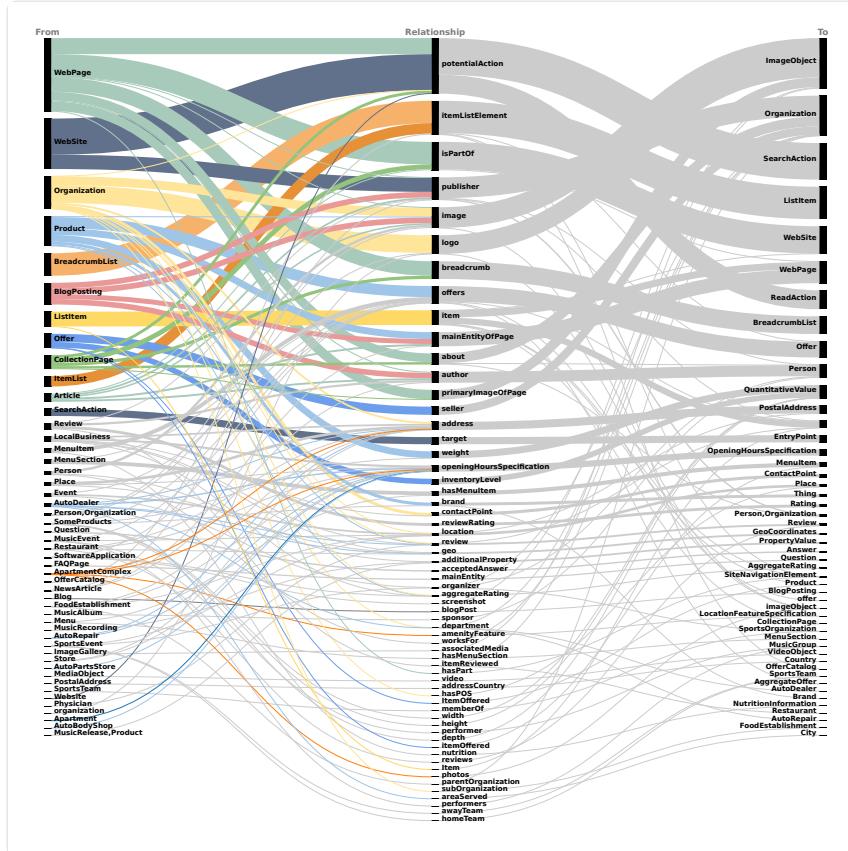


Figure 4.12. JSON-LD entity relationship as a Sankey diagram.

The diagram shows the correlation between JSON-LD entities on mobile pages and represent them as flows, visually linking entities and relationships. Each class represents a unique value in the cluster and the height is proportional to its frequency.

We're limiting in the chart the analysis to the top 200 most frequent chains.

From the chart we also get first overview of the sectors behind these graphs from general publishing to e-commerce from local business to events, automotive, music and so on.

## Relationship depth

Out of curiosity, we also calculated the deepest, most complex relationships between entities—in both our mobile and desktop data sets.

Deeper relationships *tend* to equate to richer, more comprehensive descriptions of entities (and the other entities they're related to).

# 18

Figure 4.13. Deepest nested relationship on desktop.

The deepest relationships are:

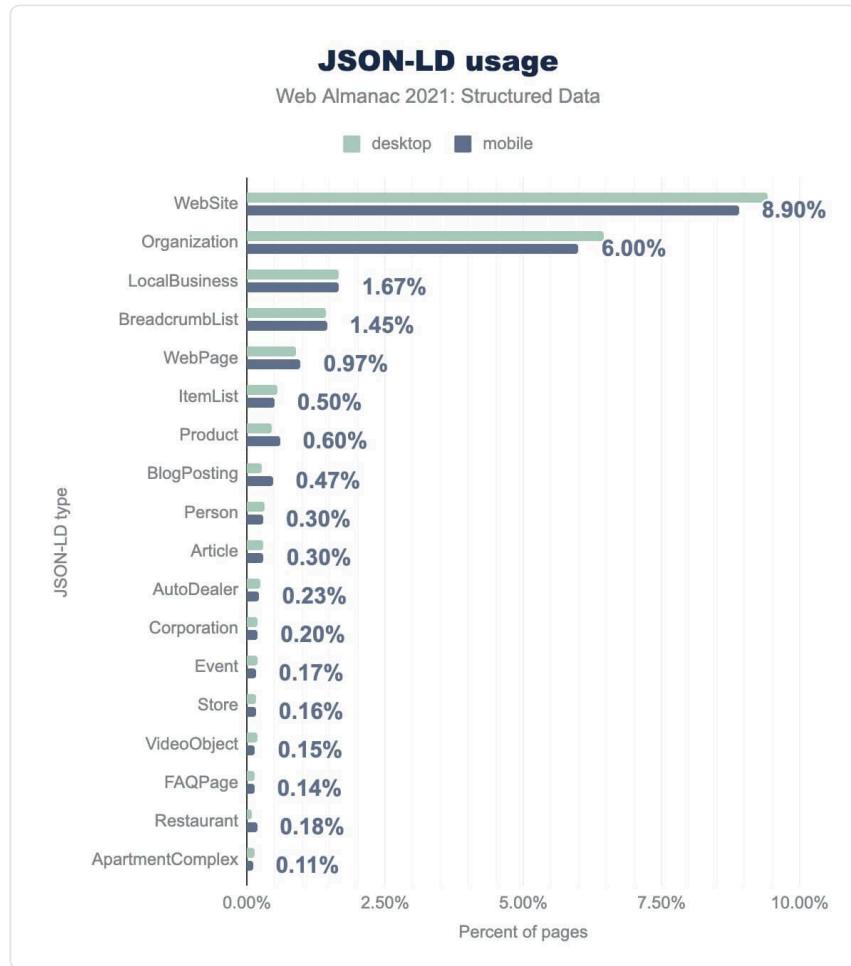
- On desktop, a depth of 18 nested connections.
- On mobile, a depth of 12 nested connections.

It's worth considering that these levels of depth may hint at programmatic generation of output, rather than hand-crafted markup, as these structures become challenging to describe and maintain at scale.

## Use of `sameAs`

One of the most powerful use-cases for structured data to declare when an entity is the `sameAs` another entity. Building a comprehensive understanding of a *thing* often requires consuming information which exists in multiple locations and formats. Having a way in which each of those instances can cross-reference the others makes it much easier to "connect the dots" and to build a richer understanding of that entity.

Because this is such a powerful tool, we've taken the time to explore some of the most common types of `sameAs` usage and relationships.

Figure 4.14. *SameAs* usage

The `sameAs` property accounts for 1.60% of all JSON-LD markup and is present on 13.03% of pages.

We can see that the most common values of the `sameAs` property (normalizing from URLs to hostnames) are social media platforms (e.g., [facebook.com](https://facebook.com), [instagram.com](https://instagram.com)), and official sources (e.g., [wikipedia.org](https://wikipedia.org), [yelp.com](https://yelp.com))—with the sum of the former accounting for ~75% of usage.

It's clear that this property is primarily used to identify the social media accounts of websites and businesses; likely motivated by Google's historical reliance on this data as an input for managing *knowledge panels* in their search results. Given that this requirement was deprecated

in 2019<sup>150</sup>, we might expect this data set to gradually alter in coming years.

## Conclusion

Structured data is used broadly, and diversely, across the web. Whilst some of this is undoubtedly stale (legacy sites/pages, using outmoded formats), there is also strong adoption of new and emerging standards.

Anecdotally, much of the adoption we see of modern standards like schema.org (particularly via JSON-LD) appears to be motivated by organizations and individuals who wish to take advantage of search engines' support (and rewards) for providing data about their pages and content. But outside of this, there's a rich landscape of people who use structured data to enrich their pages for other reasons. They describe their websites and content so that they can integrate with other systems, so that they can better understand content, or in order to facilitate others to tell *their own stories* and build their own products.

A web made of deeply connected, structured data which powers a more integrated world has long been a science-fiction dream. But perhaps, not for much longer. As these standards continue to evolve, and their adoption continues to grow, we pave a road towards an exciting future.

## Future years

In future years we hope to be able to continue the analysis started here, and to map the evolution of structured data usage over time.

We look forward to exploring further.

## Authors



### Jono Alderson

@jonoalderson jonoalderson <https://www.jonoalderson.com>

Jono Alderson is a digital strategist, marketing technologist, and full stack developer. He enjoys dabbling with website performance, technical SEO, schema.org and all things structured data.

150. <https://twitter.com/googlesearchc/status/1143558928439005184>



## Andrea Volpini

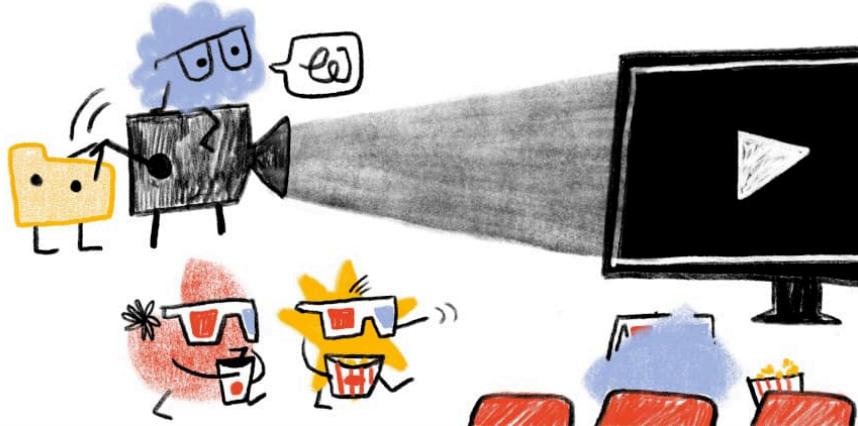
🐦 @cyberandy 🌐 cyberandy 🌐 <https://wordlift.io/blog/en/entity/andrea-volpini/>

Andrea Volpini is the CEO of WordLift, and is currently focusing on the semantic web, SEO and artificial intelligence.

---

# Part I Chapter 5

# Media



**Written by Eric Portis and Doug Sillars**

*Reviewed by Navaneeth Krishna, Tamas Piros, Akshay Ranganath, and Addy Osmani*

*Analyzed by Eric Portis, Doug Sillars, and Akshay Ranganath*

*Edited by Barry Pollard*

## Introduction

Almost three decades ago the `<img>` tag dropped and hypertext became hypermedia. The web has become increasingly visual ever since. What is the state of media on the web in 2021? Let's look at images and videos, in turn.

## Images

Images are ubiquitous on the web. Almost every page contains image content.

# 95.9%

*Figure 5.1. Percentage of pages that contained at least one contentful <img>*

And effectively all pages serve up some sort of imagery (even if it's just a background or favicon).

# 99.9%

*Figure 5.2. Percentage of pages that generated at least one request for an image resource*

The impact that all of these images have is hard to overstate. As the Page Weight chapter highlights, images are still responsible for more bytes-per-page than any other resource type. However, year-over-year, per-page image transfer sizes have decreased.



*Figure 5.3. Mobile image transfer size by year.*

This is surprising. For the last decade, the Image Bytes<sup>151</sup> chart on the HTTP Archive's monthly State of Images report<sup>152</sup> has seemingly only ever gone one direction: up. What reversed this

151. <https://httparchive.org/reports/state-of-images#bytesImg>

152. <https://httparchive.org/reports/state-of-images>

trend in 2021? I think it may have something to do with native lazy-loading's rapid adoption, which we will discuss more later in the chapter.

In any case, by quantity, images continue to make up an awful lot of the stuff of the web. But tallying the sheer number of elements, requests, and bytes doesn't tell us how crucial images are to users' experiences. To get a sense of that, we can look at the Largest Contentful Paint<sup>153</sup> metric, which tries to identify the most important piece of above-the-fold content on any given page. As you can see in the Performance chapter, the LCP element has an image on around three quarters of pages.

# 70.6%

*Figure 5.4. Mobile pages whose LCP element has an image. On the desktop it's 79.4%!*

Images are crucial to user's experiences of the web! Let's dive in, taking a closer look at how they're encoded, embedded, laid out, and delivered.

## Encoding

Image data on the web is encoded in files. What can we say out about these files, and the image data that they contain?

Let's start by looking at their pixel dimensions. We'll start small.

### Single pixel images

Many `<img>` elements don't actually represent contentful images<sup>154</sup> and instead, they contain only a single pixel:

Client 1x1 images	
Mobile	7.5%
Desktop	7.0%

*Figure 5.5. Single pixel image use.*

153. <https://web.dev/lcp/>

154. <https://www.merriam-webster.com/dictionary/image>

These single pixel `<img>` elements are, put bluntly, hacks: they are being abused either to do layout<sup>155</sup> (which would be better done with CSS) or to track users<sup>156</sup> (which would be better-accomplished using the Beacon API<sup>157</sup>).

We can establish a baseline breakdown of what jobs all of these single pixel `<img>`s are doing by looking at how many use data URIs<sup>158</sup>.



*Figure 5.6. Data URI single pixel images.*

The single pixel `<img>`s containing data URIs are almost certainly being used for layout. The remaining ~54% which generate a request might be there for layout or they might be tracking pixels—we can't tell.

Note that throughout the rest of this analysis, we have excluded single pixel `<img>`s from the results. For this media chapter, we're interested in `<img>` elements that are presenting visual information to the user, not tracking pixels or layout hacks.

## Multiple pixel images

When `<img>`s contain more than one pixel, how many pixels do they contain?

155. [https://en.wikipedia.org/wiki/Spacer\\_GIF](https://en.wikipedia.org/wiki/Spacer_GIF)

156. [https://en.wikipedia.org/wiki/Web\\_beacon](https://en.wikipedia.org/wiki/Web_beacon)

157. [https://developer.mozilla.org/docs/Web/API/Beacon\\_API](https://developer.mozilla.org/docs/Web/API/Beacon_API)

158. [https://developer.mozilla.org/docs/Web/HTTP/Basics\\_of\\_HTTP/Data\\_URIs](https://developer.mozilla.org/docs/Web/HTTP/Basics_of_HTTP/Data_URIs)

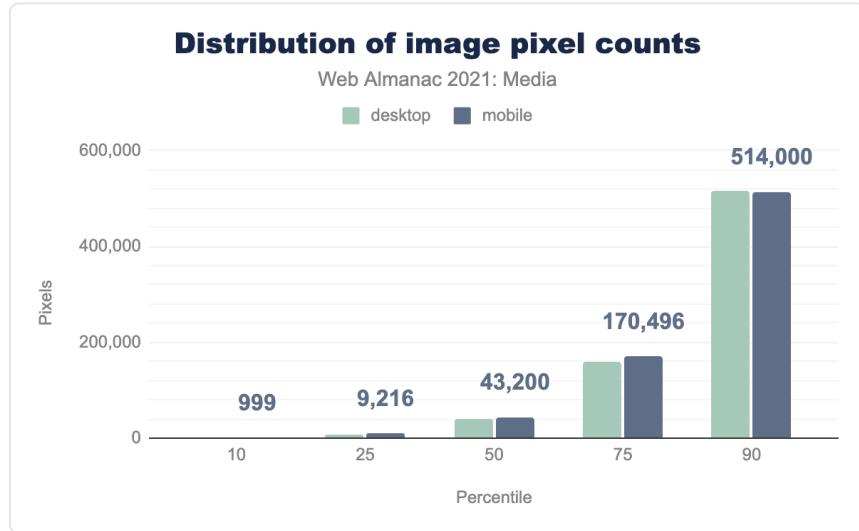


Figure 5.7. Distribution of image pixel counts.

The median `<img>` loads just over 40,000 pixels on mobile. I found this number surprisingly small. Just under half of crawled `<img>`s (excluding the ones that loaded single pixel images, or nothing at all) contain about the same number of pixels as a 200x200 image.

However, when you consider the number of `<img>` elements per page, this statistic is less surprising. Most pages contain more than 15 images, so they are often made up of many smaller images and icons. Thus, while images with more than half-a-megapixel might only account for one in ten `<img>` elements, they are not at all uncommon, as we navigate across pages. Many pages will include at least one larger image.



Figure 5.8. Number of `<img>`s per page.

I was also surprised that there was almost no difference between desktop and mobile at the top end of the pixel count distribution. Initially, this seemed to indicate a lack of effective adoption of responsive image features, but when you consider that the mobile crawler has a  $360 \times 512\text{px}$  @3x viewport (so 1,080 by 1,536 physical pixels), while the desktop viewport is  $1,376 \times 768\text{px}$  @1x, it isn't actually surprising: the crawlers' viewports had similar widths, in physical pixels (1,080 vs 1,376). A bigger difference in physical pixel resolution between the crawlers would be more revealing.

## Aspect ratios

Images on the web are mostly landscape-oriented, and portrait-oriented images are relatively rare.



Figure 5.9. Image orientations.

This feels like a missed opportunity on mobile. The success of the “stories” UI pattern<sup>159</sup> shows that there’s value in imagery tailored to fill portrait-oriented mobile screens.

Images’ aspect ratios were clustered around “standard” values, such as 4:3, 16:9, and especially 1:1 (square). The top 10 aspect ratios accounted for nearly half of all `<img>`s:

159. <https://uxdesign.cc/the-powerful-interaction-design-of-instagram-stories-47cdeb30e5b6>

Aspect ratio	Desktop images	Mobile images
1:1	32.9%	32.7%
4:3	3.7%	4.1%
3:2	2.5%	2.6%
2:1	1.6%	1.7%
16:9	1.5%	1.5%
3:4	0.9%	1.0%
2:3	0.7%	0.7%
5:3	0.6%	0.5%
6:5	0.5%	0.5%
8:5	0.5%	0.5%

Figure 5.10. A ranked list of the top ten image aspect ratios.

## Bytes

Let us turn our attention to file sizes.



Figure 5.11. Distribution of image byte sizes.

The median contentful `<img>` weighs in at just over 10kB. But, again, the median page contains more than 15 `<img>`s so, when looking at the ninetieth percentile of all images across pages, images that push past 100kB aren't rare at all.

## Bits per pixel

Bytes and dimensions are interesting on their own, but to get a sense of how compressed the web's image data is, we need to put bytes and pixels together, to calculate *bits per pixel*. Doing so allows us to make apples-to-apples comparisons of the information density of images, even if those images have different resolutions.

In general, bitmaps on the web decode to eight bits of information per channel, per pixel. So, if we have an RGB image with no transparency, we can expect a decoded, uncompressed image to weigh in at 24 bits per pixel<sup>160</sup>. A good rule of thumb for *lossless* compression is that it should reduce file sizes by a 2:1 ratio (which would work out to 12 bits per pixel for our 8-bit RGB image). The rule of thumb for 1990s-era lossy compression schemes (JPEG and MP3) was a 10:1 ratio (2.4 bits/pixel). It should be noted that, depending on image content and encoding settings, these ratios vary *widely*, and modern JPEG encoders like MozJPEG<sup>161</sup> typically outperform this 10:1 target at their default settings.

<sup>160</sup>. [https://en.wikipedia.org/wiki/Color\\_depth#True\\_color\\_\(24-bit\)](https://en.wikipedia.org/wiki/Color_depth#True_color_(24-bit))

<sup>161</sup>. <https://github.com/mozilla/mozjpeg>

So, with all of that context, here's how the web's images stack up:



Figure 5.12. Distribution of image bits per pixel.

The median `<img>` on mobile hits that 10:1 compression ratio target on the nose: 2.4 bits/pixel. However, around that median, there is a tremendous spread. Let's break things down by format in order to learn a bit more.

## Bits per pixel, by format



Figure 5.13. Median bits per pixel by format.

The median JPEG weighs in at 2.1 bits per pixel. Given the format's ubiquity, this is the best baseline to measure other formats by.

The median PNG weighs in at more than twice that. PNG is sometimes called a *lossless* format, but a median of 4.6 bits per pixel shows how false this is. True lossless compression should typically land at around 12-16 bits per pixel (depending on whether or not we're dealing with an alpha channel). PNG comes in so far below this because common PNG tooling is usually *lossy*: it modifies pixels—reducing color palettes and introducing dithering patterns—before encoding pixels, to boost compression ratios.

GIFs, weighing in at 7.4 bits per pixel, come off terribly here, and make no mistake, they<sup>162</sup> are<sup>163</sup> terrible!<sup>164</sup>! But they're also at a bit of an unfair disadvantage here because many GIFs on the web are animated. Web platform APIs don't expose the number of frames in an animated image, so we haven't accounted for frames. To give you a sense of how much this inflates GIF's numbers: a GIF measured as 20 bits per pixel, here, which contains ten frames, should be fairly counted as using two bits per pixel.

Things get really interesting when we look at two next-gen formats: WebP and AVIF. Both weigh in almost 40% lighter than JPEG, at 1.3-1.5 bits per pixel. In formal studies using matched

162. <https://web.dev/efficient-animated-content/>

163. <https://bits-of-code.de/optimising-gifs/>

164. <https://dougsillars.com/2019/01/15/state-of-the-web-animated-gifs/>

qualities<sup>165</sup>, WebP outperforms JPEG by between 25-34%, so its real-world performance seems surprisingly good. On the other hand, AVIF's creators have published data suggesting that it is capable of outperforming modern JPEG encoders JPEG by 50%+, in the lab<sup>167</sup>. So, while AVIF's performance here is good, I expected it to be better. I can think of a few possible explanations for these discrepancies between lab data and real-world performance.

First: tooling. JPEG encoders vary incredibly widely, ranging from hardware encoders in cameras which don't spend much effort compressing images well, to ancient copies of `libjpeg` installed decades ago, to bleeding-edge, best-practice-by-default encoders like MozJPEG. In short, there are a lot of old, badly compressed JPEGs out there, but every WebP and AVIF has been compressed with modern tooling.

Also, anecdotally, the reference WebP encoder (`cwebp`) is relatively aggressive about quality/compression, and returns lower-quality, more-compressed results by default than most common JPEG tooling.

As far as AVIF is concerned: `libavif` is capable of a wide variety of compression ratios depending on which "speed" setting you choose. At its slowest speeds (producing the highest-efficiency files) `libavif` can take minutes to encode a single image. It's reasonable to assume that different image-rendering pipelines will make different tradeoffs when choosing speed settings, depending on their constraints. This results in a wide distribution of compression performance.

Another thing to keep in mind when evaluating AVIF's real-world performance here, is that there just aren't that many AVIFs on the web, yet. The format is currently being used by relatively few sites, on a limited set of content, so we don't yet have a full sense of how it will ultimately perform "in the wild." This will be interesting to track over the coming years, as adoption increases (and tooling improves).

One thing that is absolutely clear is that both WebP and AVIF can be used to deliver a wide variety of content (including photography, illustrations<sup>168</sup>, and images with transparency) more efficiently than the web's legacy formats. But, as we'll see in the next section, not that many sites have adopted them.

165. <https://kornel.ski/en/faircomparison>

166. [https://developers.google.com/webp/docs/webp\\_study](https://developers.google.com/webp/docs/webp_study)

167. <https://netflixtechblog.com/avif-for-next-generation-image-coding-b1d75675fe4>

168. <https://jakearchibald.com/2020/avif-has-landed/#flat-illustration>

## Format adoption



Figure 5.14. Image format adoption (mobile).

The old formats still reign: JPEG dominates, with PNG and GIF rounding out the podium. Together, they account for almost 90% of the images on the web. WebP—which is now more than a decade old but which only achieved universal browser support last year<sup>169</sup>—is still in the single digits. And effectively no-one is using AVIF, which accounted for only 0.04% of crawled resources. We found a thousand JPEGs for every AVIF.

For an in-depth analysis of how (and educated guesses as to why) WebP and AVIF adoption has changed over time, the best resource is Paul Calvano<sup>170</sup>'s excellent recent talk at ImageReady (full video<sup>171</sup> and slides 13-15<sup>172</sup>). In it, he shows that WebP adoption increased by ~34% from July 2020 (when Safari added support) to July 2021. AVIF's numbers have risen even more rapidly, in percentage terms, though perhaps that's not surprising given that the format is still brand new and used by relatively few sites. A few large<sup>173</sup> players<sup>174</sup> adopting AVIF was all it took.

## Embedding

In order to display an image on a web page, we must embed it, using the `<img>` element. This

169. <https://www.macrumors.com/2020/06/22/webp-safari-14/>

170. <https://twitter.com/paulcalvano>

171. <https://www.youtube.com/watch?v=tz5bpAQY43k>

172. [https://docs.google.com/presentation/d/1VSQJNR6lh2y9jL5xaeainQ2cTAWyy7QIEJDMh4hNQA/edit#slide=id.gfc0d6ffce\\_0\\_0](https://docs.google.com/presentation/d/1VSQJNR6lh2y9jL5xaeainQ2cTAWyy7QIEJDMh4hNQA/edit#slide=id.gfc0d6ffce_0_0)

173. <https://twitter.com/chriscoyier/status/1465474900588646408>

174. <https://medium.com/vimeo-engineering-blog/upgrading-images-on-vimeo-620f79da8605>

venerable element has gained a handful of new features over the past few years but how are those features being put into practice?

## Lazy-loading

If there is one breakout story this year as far as images on the web, it is native lazy-loading<sup>175</sup> adoption. Look at this chart:

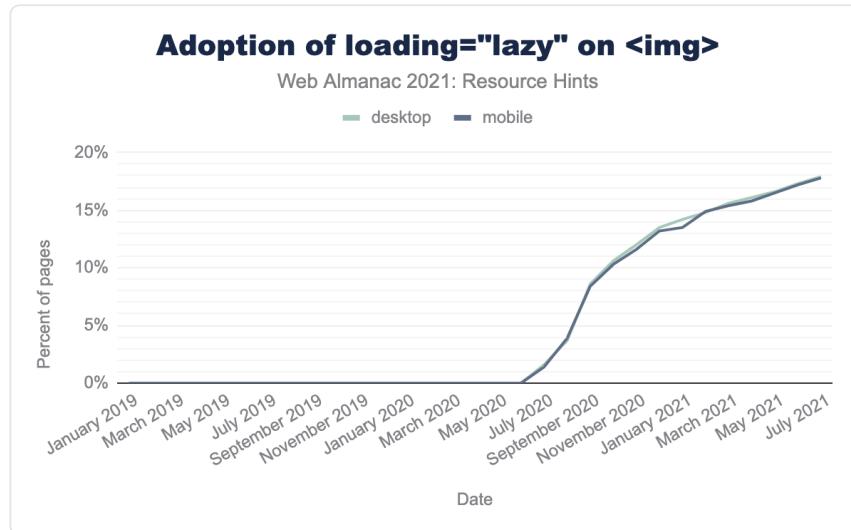


Figure 5.15. Adoption of `loading="lazy"` on `<img>`.

In July of 2020, native lazy-loading was used on just 1% of pages. By July of 2021, that number had exploded, to 18%. This is an unbelievable rate of growth considering the vast number of pages and templates which are not updated from year to year.

Personally, I think native lazy-loading's rapid adoption is the best explanation we have for the trend-breaking reduction in image bytes per page, this year.

What fueled lazy-load adoption? There's some consensus that it was a combination of ease of use, pent-up developer demand, and WordPress enabling lazy-loading by default across a vast swath of the web<sup>176</sup>.

Perhaps native lazy-loading has been too successful? The Resource Hints chapter notes that the majority of lazy-loaded images were in the initial viewport (whereas the feature is ideally

175. <https://web.dev/browser-level-image-lazy-loading/>

176. <https://make.wordpress.org/core/2020/07/14/lazy-loading-images-in-5-5/>

used on “below the fold” images). Furthermore, the Performance chapter found that 9.3% of Largest Contentful Paint elements have their `loading` attribute set to `lazy`, which significantly delays the page’s most important piece of content from loading, and hurts users’ experiences.

## Decoding

The `decoding` attribute on `<img>` serves as a useful point of contrast to highlight native lazy-loading’s success. First supported<sup>177</sup> in 2018—about a year before native lazy-loading—the `decoding` attribute allows developers to prevent large image decode operations from blocking the main thread. It provides functionality that not all web developers need or understand, and that shows in the usage data. `decoding` is used on just 1% of pages, and on only 0.3% of `<img>` elements.

## Accessibility

When you embed contentful images on web pages, you should make their content as accessible as possible for non-visual users. I note this only to refer you to the Accessibility chapter, whose in-depth analysis of image accessibility on the web found small year-over-year progress, but mostly: a whole lot of room for improvement.

## Responsive images

In 2013, a suite of features enabling adaptive image loading on responsive websites landed, too much fanfare. Eight years in, how are responsive image features being used?

First, let us consider the `srcset` attribute, which allows developers to supply multiple possible resources for the same `<img>`.

### `x` and `w` descriptor adoption

# 30.9%

Figure 5.16. Percent of mobile pages that use `srcset`.

Almost a third of crawled pages use `srcset`—pretty good!

177. <https://www.chromestatus.com/feature/4897260684967936>

And `w` descriptors, which allow browsers to select a resource based on both varying layout widths and varying screen densities<sup>178</sup>, are four times more popular than `x` descriptors, which only enable DPR-adaptation<sup>179</sup>.

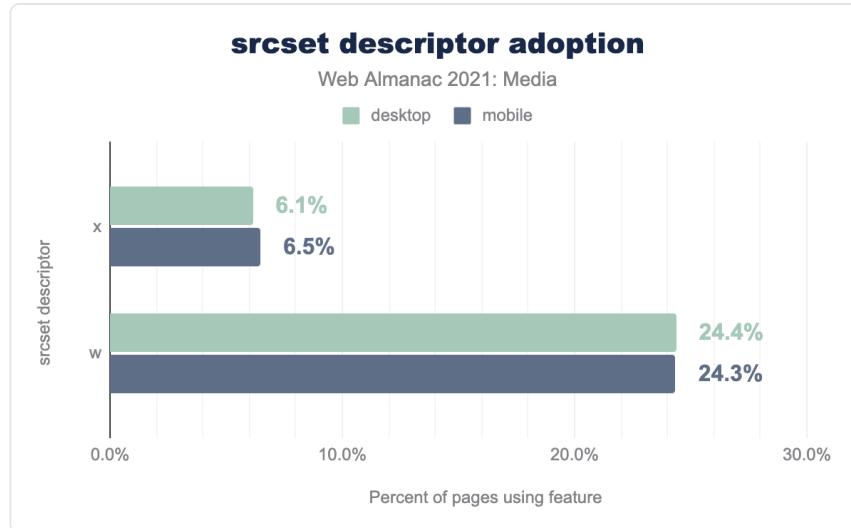


Figure 5.17. `srcset` descriptor adoption.

How are developers populating their `srcset`s with resources?

### Number of `srcset` candidates

Let's first take a look at the number of candidate resources developers are including:

178. <https://jakearchibald.com/2015/anatomy-of-responsive-images/#varying-size-and-density>  
 179. <https://jakearchibald.com/2015/anatomy-of-responsive-images/#fixed-size-varying-density>



Figure 5.18. Number of srcset candidates.

A large majority of `srcset`s are populated with five-or-fewer resources.

### `srcset` density ranges

Are developers giving browsers an appropriately wide range of choices, within their `srcset`s?

In order to answer this question, we must first understand how `srcset` and `sizes` values are used by browsers.

When browsers pick a resource to load out of a `srcset`, they first assign every candidate resource a density<sup>180</sup>. Calculating the density of resources that use `x` descriptors is straightforward. A resource with a `2x` density descriptor has a density of (wait for it) `2x`.

`w` descriptors complicate things. What's the density of a `1000w` resource? It depends on the resolved `sizes` value (which might depend on the viewport width!). When `w` descriptors are used, each descriptor is divided by the resolved `sizes` value, to determine its density. For example:

```
<img  
srcset="large.jpg 1000w, medium.jpg 750w, small.jpg 500w"
```

180. <https://html.spec.whatwg.org/multipage/images.html#current-pixel-density>

```
sizes="100vw"
/>>
```

On a 500-CSS-px-wide viewport, these resources will be assigned the following densities:

Resource	Density
large.jpg	$1000w \div 500px = 2x$
medium.jpg	$750w \div 500px = 1.5x$
small.jpg	$500w \div 500px = 1x$

However, on a 1000-CSS-px-wide viewport, these same resources, marked up with the same `srcset` and `sizes` values, will have different densities:

Resource	Density
large.jpg	$1000w \div 1000px = 1x$
medium.jpg	$750w \div 1000px = 0.75x$
small.jpg	$500w \div 1000px = 0.5x$

After these densities are calculated, browsers pick the resource with the density that's the best match for the current browsing context. It's safe to say that in this example, the `srcset` did not contain a wide-enough range of resources. Viewports measuring more than 1,000 CSS px across, with higher than `1x` densities, are not uncommon; if you're reading this on a laptop, you're probably browsing in such a context, right now. And in these contexts, the best browsers can do is pick `large.jpg`, whose `1x` density will still appear blurry on the high-density display.

So, armed with both:

1. an understanding of how browsers turn `x` and `w` descriptors, `sizes` values, and browsing contexts into resource densities.
2. an understanding of how the range of resource densities in a `srcset` changes across browsing contexts, and ultimately impacts users.

...let's look at the ranges of densities supplied by the `srcset`s that use either `x` descriptors or `w` descriptors:



Figure 5.19. Ranges of densities covered by `srcset`s that use either `x` or `w` descriptors.

As you interpret this data, keep in mind the viewports of the two different crawlers:

- Desktop: 1,376 × 768px @1x
- Mobile: 360 × 512px @3x

Different viewport widths would have altered many resolved `sizes` values and given different results.

That said, these results look good. Nine out of ten `srcset`s are providing a range of resources that covers a reasonable range of output display densities (1x-2x), even on the larger desktop viewport. Given the exponential bandwidth costs and diminishing visual returns of densities above 2x<sup>181</sup>, the steep drop-off after 2x seems not only reasonable, but perhaps even optimal.

### `sizes` accuracy

Responsive images can be tricky. Authoring reasonably-accurate `sizes` attributes—and keeping them up to date with evolving page layouts and content—might be the hardest part about getting responsive images right. How many authors get `sizes` wrong? And how wrong do they get it?

<sup>181</sup>. [https://blog.twitter.com/engineering/en\\_us/topics/infrastructure/2019/capping-image-fidelity-on-ultra-high-resolution-devices](https://blog.twitter.com/engineering/en_us/topics/infrastructure/2019/capping-image-fidelity-on-ultra-high-resolution-devices)



Figure 5.20. Distribution of <img> sizes errors.

More than a quarter of `sizes` attributes are perfect: exact matches for the layout size of the image. As someone who has authored a number of erroneous `sizes` attributes, myself, I found this both surprising and impressive. That is, until I realized that the accuracy measurement here was taken after JavaScript runs, and many `sizes` attributes are ultimately written by client-side JavaScript. Here are the most common `sizes` values, before JavaScript runs:

<b>Sizes</b>	<b>Desktop</b>	<b>Mobile</b>
<code>auto</code>	8.2%	9.6%
<code>(max-width: 300px) 100vw, 300px</code>	4.7%	5.9%
<code>(max-width: 150px) 100vw, 150px</code>	1.3%	1.6%
<code>(max-width: 600px) 100vw, 600px</code>	1.0%	1.2%
<code>(max-width: 400px) 100vw, 400px</code>	1.0%	1.1%
<code>(max-width: 800px) 100vw, 800px</code>	0.8%	0.9%
<code>(max-width: 500px) 100vw, 500px</code>	0.8%	0.9%
<code>(max-width: 1024px) 100vw, 1024px</code>	0.7%	0.9%
<code>(max-width: 320px) 100vw, 320px</code>	0.5%	0.8%
<code>(max-width: 100px) 100vw, 100px</code>	0.7%	0.8%
<code>100vw</code>	0.7%	0.7%

Figure 5.21. A ranked list of the most common sizes attribute values, before JavaScript runs.

One in ten `sizes` attributes on mobile has an initial value of `auto`. This non-standard value is then presumably replaced by a JavaScript library (probably `lazysizes.js`<sup>182</sup>), using the measured layout size of the image.

Some error in `sizes` is acceptable as the attribute provides a pre-layout hint to the browser in order to help it select an appropriate resource to load, before layout is complete. But large errors can lead to bad resource choices. This appears likely for the least-accurate quarter of `sizes` attributes, which report widths twice as large as the actual `<img>` layout width on desktop and 1.5x as large as the actual `<img>` layout width on mobile.

So: one in ten `sizes` attributes is being authored on the client by a JavaScript library, and at least one in four is inaccurate enough that the error is likely to impact resource selection. Both of these facts represent significant opportunities for either existing tooling<sup>183</sup> or new web platform features<sup>184</sup> to help more authors get `sizes` right.

182. <https://github.com/aFarkas/lazysizes>

183. <https://github.com/ausi/respimgalint>

184. <https://github.com/whatwg/html/issues/4654>

## <picture> usage

The `<picture>` element serves a couple of use cases:

1. Art direction, with the `media` attribute
2. Format-switching, based on MIME-type, via the `type` attribute

# 5.9%

*Figure 5.22. The percentage of mobile pages which use `<picture>`.*

`<picture>` is used much less frequently than `srcset`. Here's how usage breaks down between those two use cases:



*Figure 5.23. <picture> feature usage.*

Art direction appears a bit more popular than format-switching, but both features appear underutilized when you consider their potential utility. As we've seen, very few pages are tailoring images' aspect ratios to fit mobile screens, and many more pages could deliver their imagery more efficiently using next-generation formats. These are exactly the problems that `<picture>` was invented to solve, and perhaps more than 5.9% of pages could be addressing them, using it.

It's possible that format-switching with `<source type>` is only used on 2-3% of pages because format-switching can also be accomplished using server-side content negotiation<sup>185</sup>. Unfortunately, server-side adaptation mechanisms are hard to detect in the crawled data, and we have not analyzed them here.

Notably, `<source type>` and `<source media>` are not mutually exclusive, and, put together, the usage percentages here do not add up to 100%. This suggests that at least 15% of `<picture>` elements do not leverage either of these attributes, making those `<picture>`s functionally equivalent to a `<span>`.

## Layout

Once you've embedded an image on a page, you must lay it out amongst the rest of the page's contents. There are many, many ways to do this, but we can derive a few insights about how it's generally done by zooming out and answering a couple of big-picture questions.

### Intrinsic vs extrinsic sizing

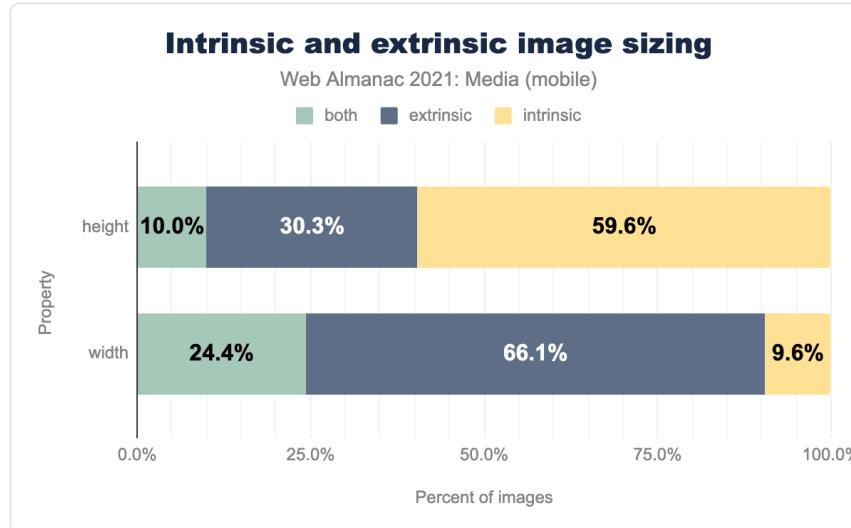
As replaced elements<sup>186</sup>, images have a natural, "intrinsic" size<sup>187</sup>. This is the size that they will render at by default, if there are no CSS rules placing "extrinsic" layout constraints upon them.

How many images are intrinsically vs extrinsically sized?

185. [https://developer.mozilla.org/docs/Web/HTTP/Content\\_negotiation](https://developer.mozilla.org/docs/Web/HTTP/Content_negotiation)

186. [https://developer.mozilla.org/docs/Web/CSS/Replaced\\_element](https://developer.mozilla.org/docs/Web/CSS/Replaced_element)

187. [https://developer.mozilla.org/docs/Glossary/Intrinsic\\_Size](https://developer.mozilla.org/docs/Glossary/Intrinsic_Size)



*Figure 5.24. Intrinsic and extrinsic image sizing.*

The question is a little complicated because some images (those with a `max-width`, `max-height`, `min-width`, or `min-height` constraint), are sometimes extrinsically sized, but sometimes left to their intrinsic size. We've labelled those images as "both."

In any case, perhaps unsurprisingly, most images have extrinsic width constraints and height-constrained sizing is much less common.

### Reducing layout shifts with `height` and `width`

This brings us to the last web platform feature that we'd like to investigate: using the `height` and `width` attributes to reserve layout space for flexible images.

By default, images left to their intrinsic dimensions take up no space until they load, and their intrinsic dimensions become known. At that point—poof—they pop into the page, causing a layout shift<sup>188</sup>. This was exactly the problem that the `height` and `width` attributes were invented to solve—in 1996<sup>189</sup>.

Unfortunately, `height` and `width` never played well with images that are assigned a variable extrinsic size in one dimension (e.g., `width: 100%;`), and left to fill out their intrinsic aspect ratio, in the other dimension. This is the dominant pattern in responsive design. So `width` and

188. <https://developers.google.com/publisher-tag/guides/minimize-layout-shift>

189. <https://www.w3.org/TR/2018/SPSD-html32-20180315/#img>

`height` fell out of favor within responsive contexts until 2019, when a tweak to how `height` and `width` are used by browsers fixed this problem. Now, consistently setting `height` and `width` is one of the best things authors can do to reduce Cumulative Layout Shift<sup>190</sup>. How often are these attributes accomplishing this task?



Figure 5.25. The percentage of `<img>`s on mobile that have both `height` and `width` attributes and are extrinsically sized in only one dimension.

It's hard to tell how many of these `<img>`s were authored with the new browser behavior in mind, but they're all benefiting from it. And that was the point—by re-using existing attributes, lots of existing content benefited from the change, automatically.

## Delivery

Finally, let's take a look at how images are delivered over the network.

### Cross-origin image hosts

How many images are being hosted by the same origin that they're being embedded on? The slimmest of minorities:

<sup>190.</sup> <https://web.dev/cls/>

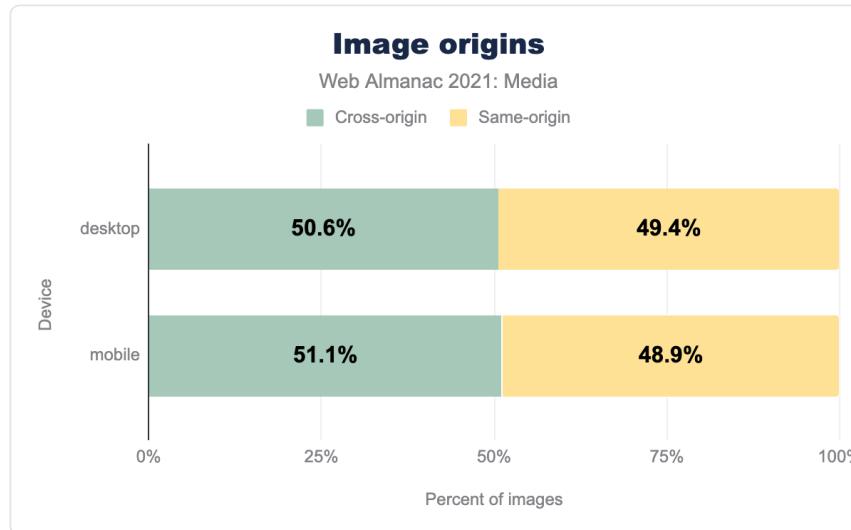


Figure 5.26. Image origins.

Cross-origin images are subject to significant security restrictions<sup>191</sup>, and can sometimes incur performance costs<sup>192</sup>. On the other hand, moving static assets to a dedicated CDN is one of the most impactful things you can do to help Time to First Byte<sup>193</sup>, and image CDNs provide powerful transformation and optimization<sup>194</sup> features which can automate all sorts of best-practices. It would be fascinating to see how many of the 51% of cross-origin images are hosted on image CDNs and to compare their performance against the rest of the web's. Unfortunately, that was outside the scope of our analysis.

And with that, it is time to turn our attention to...

## Video

As the world has dramatically changed over the last year, we have seen a huge growth in video usage on the web. In the 2020 media report, it was estimated that 1-2% of websites had a `<video>` tag. In 2021, that number has jumped drastically, with over 5% of desktop sites and 4% of mobile sites incorporating a `<video>` tag.

191. [https://developer.mozilla.org/docs/Web/HTML/CORS\\_enabled\\_image](https://developer.mozilla.org/docs/Web/HTML/CORS_enabled_image)

192. <https://andydavies.me/blog/2019/03/22/improving-perceived-performance-with-a-link-rel-equals-preconnect-http-header/>

193. [https://developer.mozilla.org/docs/Glossary/time\\_to\\_first\\_byte](https://developer.mozilla.org/docs/Glossary/time_to_first_byte)

194. <https://web.dev/image-cdns/>

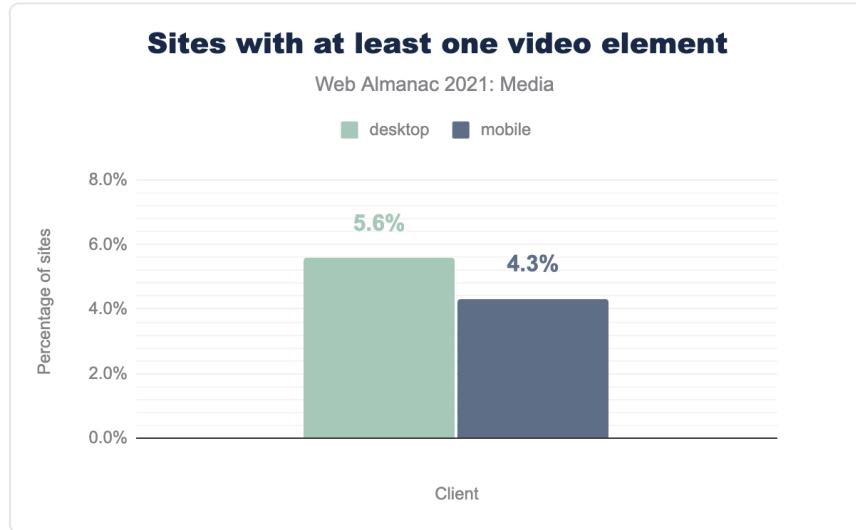
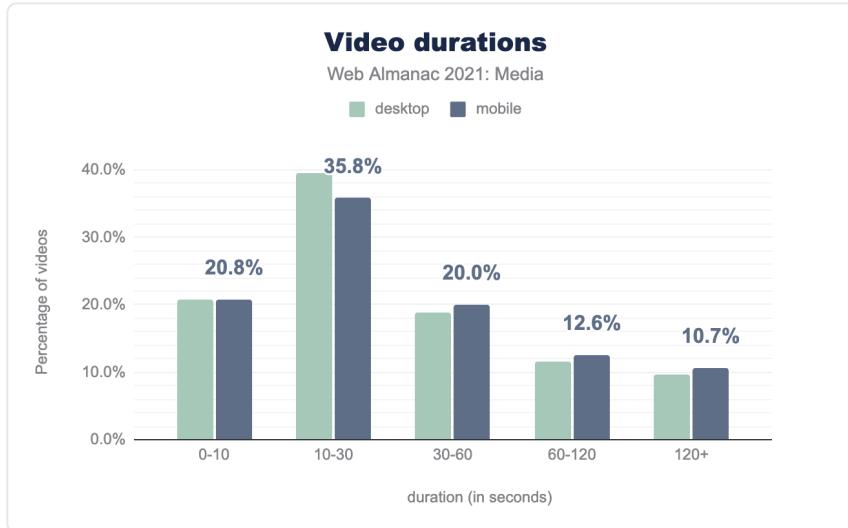


Figure 5.27. Sites with at least one video element.

This huge growth in video usage on the web indicates that as devices/networks improve, there is a desire to add immersive experiences such as video to sites.

When it comes to interaction with video, it is interesting to see how long the videos are when posted on a web page. We were able to query this value for 440k desktop videos, and 382k mobile videos, and broke down the duration into buckets of varying duration (in seconds):



*Figure 5.28. Video durations.*

Most videos on the web are short: ~ 60% of videos are under 30 seconds long on both mobile and desktop. However, 12-13% are between one and two minutes, and 10% of videos are over two minutes long.

## Video: formats

What types of files are being delivered as video? We queried all files with `video` in the MIME type, and then sorted by the file extension.

The chart below shows all video extensions with over 1% market share:



Figure 5.29. Top extensions of files with a video MIME type.

By far, the #1 video format on the web is the `mp4` (or MPEG-4), since the mp4 h264 format has 98.4% support in all modern browsers<sup>195</sup>, and the 1.9% of browsers that do not support `mp4` have no video support, so the number is really 100% coverage. Interestingly, the `mp4` usage has dropped by ~15% YOY on both desktop and mobile. WebM support also dropped significantly YOY<sup>196</sup> (50% drop on both mobile and desktop).

Where we see the growth are files with no extension (these are often from YouTube or other streaming platforms), and in web streaming. `ts` files are segments used in HTTP Live Streaming (HLS) where we see a 4% jump in usage. `.m4s` are MPEG Dynamic Adaptive Streaming over HTTP (MPEG-DASH) video segments. M4S files grew by 50% from 2.3% to 3.3% YOY.

## Video CSS: `display`

To begin, let's look at how a video will appear on a page by looking at the CSS `display` property for that video. What we find is that approximately half of all videos use a display value of `block`—placing the video on its own line and allowing for height and width values to be set for the video. The `inline-block` value also allows height and width to be specified—for a total of two thirds of all videos.

195. <https://caniuse.com/mpeg4>

196. <https://almanac.httparchive.org/en/2020/media#videos>

The `display: none` declaration hides the video from the viewer. One in five videos on the web is hidden behind this display value. From a data usage perspective, this is less than optimal, as the video is still downloaded by the browser.

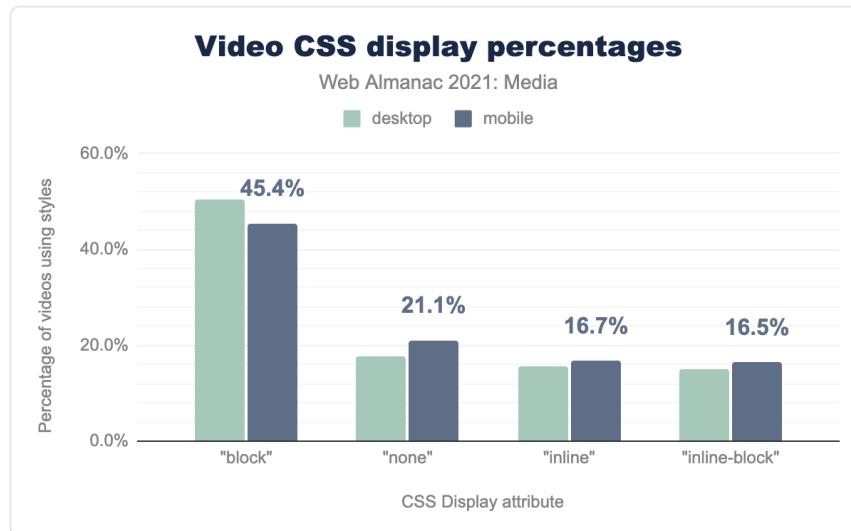


Figure 5.30. Video CSS display percentages.

## Video attributes

The `<video>` HTML5 tag has a number of attributes that can be used to define how the video player will appear to end users.

Let's look at the most common attributes and how they are used inside the `<video>` tag:

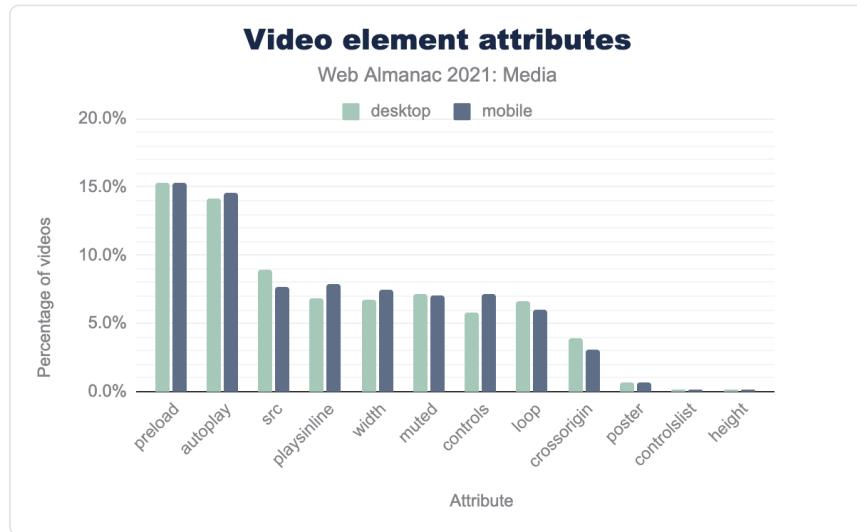


Figure 5.31. Video element attributes.

### preload

The most commonly used attribute is `preload`. The `preload` attribute gives the browser a hint on the best way to handle the video download. There are four possible options: `auto`, `metadata`, `none`, and an empty response (which uses the default of `auto`).

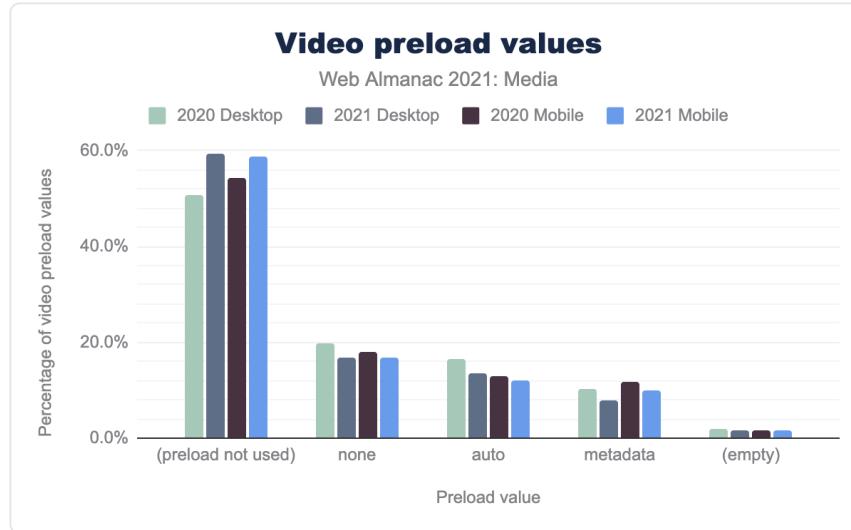


Figure 5.32. Video preload values.

Interestingly, we see a large push away from `preload` on both mobile and desktop. While it is possible that this changed for many videos, it could just be that the new videos added to the web over the last year do not utilize this setting. From a page weight perspective this is a large win for the web.

### `autoplay`

The next most commonly used attribute is `autoplay`. This tells the browser that the video should play as soon as possible (and because of this, `autoplay` will actually override the `preload` attribute).

The `autoplay` attribute is a Boolean attribute, meaning that its presence by default means true. So, for the 190 sites that use `autoplay="false"`, we're sorry to tell you that is not going to work.

### `width`

The `width` attribute is also one of the top `<video>` attributes. It tells the browser how wide the video player should be. Note that `height` is very rarely used, since the browser can set this - but it will use a default aspect-ratio of 2:1<sup>197</sup> which may be incorrect if not explicitly

197. <https://github.com/whatwg/html/issues/3090>

overridden with the `aspect-ratio` CSS styling.

The width can be presented as a percentage, or a width in pixels.

- When a percentage width is defined, the value `100%` is used in 99% of cases.
- When a width in pixels is defined, we see very similar numbers of videos at lower widths, but a large drop-off in the 1800 and 1920 widths:



Figure 5.33. Video widths.

It appears that about half of sites with larger videos that also define the width of the video remove the larger videos for mobile devices. Since very few devices need a 1080p (1920 wide) video embedded in a website, this makes sense.

### `src` and `<source>`

The `src` attribute is used in the `<video>` tag to point to the URL of the video to be played. Another way to reference the video is to use the `<source>` element.

One of the key ideas behind the `<source>` element is that the developer can supply multiple video formats to the browser, and the browser will select the first format that the browser understands.

When we look at `<source>` usage, we see that about 40% of videos have no `<source>`

element—implying that they use the `src` attribute. This is similar to the ratio found in 2020 (35%).



Figure 5.34. `source` element count.

We also see that the `<source>` element is most often used with just one element (50% of all `<video>` tags). Only 10% of `<video>` elements have 2 or more video sources named. By a 3:1 ratio, 2 is more common than 3 sources, and then there is a small selection of more than 3 (there is one video with 48 sources!).

Let's look at the videos that use 2 sources. Here are the top 10 occurrences:

Format	Desktop	Mobile
["video/mp4", "video/webm"]	25.9%	26.1%
["video/webm", "video/mp4"]	22.3%	23.3%
["video/mp4", "video/ogg"]	20.2%	24.2%
[null, null]	14.1%	8.0%
["video/mp4"]	3.6%	3.4%
["video/mp4", "video/mp4"]	3.5%	5.1%
["application/x-mpegURL", "video/mp4"]	2.4%	2.1%
[]	2.1%	1.8%
["video/mp4; codecs='avc1.42E01E, mp4a.40.2", "video/webm; codecs='vp8, vorbis']	0.8%	0.3%
["video/mp4;", "video/webm;"]	0.4%	0.3%

Figure 5.35. The most common ordered pairs of `type` values, when there are two `source` elements within a `video` element.

In six of the top 10 examples, the MP4 is listed as the first source. MP4 support on the web is at 98.4%<sup>198</sup>, and the browsers that do not support MP4 generally do not support the `<video>` tag at all. This implies that these sites do not need two sources and could save some storage on their web servers by removing their WebM or Ogg video sources—or they could reverse the order of the videos, and browsers that support WebM will download the WebM.

The same trend holds for `<video>` elements with three sources—eight of the top 10 examples begin with MP4.

198. <https://caniuse.com/mpeg4>

<b>Format</b>	<b>Desktop</b>	<b>Mobile</b>
["video/mp4", "video/webm", "video/ogg"]	30.4%	28.6%
["video/mp4; codecs=avc1", "video/mp4; codecs=avc1", "video/mp4; codecs=avc1"]	13.3%	16.4%
["video/webm", "video/mp4", "video/ogg"]	7.0%	6.3%
["video/mp4; codecs=avc1"]	5.8%	7.1%
["video/mp4", "video/ogg", "video/webm"]	5.0%	5.5%
["video/mp4;", "video/ogg; codecs="theora, vorbis", "video/webm; codecs="vp8, vorbis"]	3.8%	1.2%
["video/mp4; codecs=hevc", "video/webm", "video/mp4"]	3.2%	3.4%
["video/mp4"]	3.0%	3.0%
["video/ogg; codecs="theora, vorbis", "video/webm", "video/mp4"]	2.7%	3.3%
["video/mp4", "video/webm", "video/ogg"]	2.5%	1.7%

Figure 5.36. The most common ordered triplets of `type` values, when there are three `source` elements within a `video` element.

Of course, these implementations will just play the MP4 file, and the WebM and Ogg files will be ignored.

The incorporation of video on the web has grown immensely over the last year—jumping from 1-2% of web pages to 4-5%. We expect this growth to continue. Interestingly, the “king of video”, MP4, while still the king, is having its market share eroded by video streaming formats (that feature responsive and adaptive video sizing).

We do see movement to more performant usage of the `<video>` tag—with less use of `preload=auto`—and more use of `preload=None` as well as we see behaviors in the `width` attribute that indicate that videos are being modified (or removed) for smaller screens.

# Conclusion

As we stated at the outset: the web is increasingly visual, and the ways in which we use the web's evolving feature set to encode, embed, lay out, and deliver media continue to evolve. This year, native lazy-loading stemmed the tide of ever-increasing image transfer sizes. And universal support for WebP and initial support for AVIF pave the way for a visually richer and more efficient future. On the video side, we saw an explosion in the number of `<video>` elements and increasing use of sophisticated delivery methods like adaptive bitrate streaming.

The Web Almanac is a chance to take stock and look back. It's also a time to chart a path forward. Here's to ever-more effective visual communication on the web in 2022.

## Authors



### Eric Portis

👤 eeeeps 🌐 <https://ericportis.com/>

Eric Portis is a Web Platform Advocate at Cloudinary<sup>199</sup>.



### Doug Sillars

👤 dougsillars

Doug Sillars is a leader in developer relations, and a digital nomad working on the intersection of performance and media. He tweets @dougsillars, and blogs regularly at [dougsillars.com](https://dougsillars.com)<sup>200</sup>.

199. <https://cloudinary.com/>

200. <https://dougsillars.com>



# Part I Chapter 6

# WebAssembly



**Written by Ingvar Stepanyan**

*Reviewed by Jarrod Overson, Carlo Piovesan, Alon Zakai, Rick Viscomi, and Barry Pollard*

*Analyzed by Ingvar Stepanyan*

*Edited by Shaina Hantsis*

## Introduction

WebAssembly<sup>201</sup> is a binary instruction format that allows developers to compile code written in languages other than JavaScript and bring it to the web in an efficient, portable package. The existing use-cases range from reusable libraries and codecs to full GUI applications. It's been available in all browsers since 2017—for 4 years now—and has been gaining adoption since, and this year we've decided it's a good time to start tracking its usage in the Web Almanac.

## Methodology

For our analysis we've selected all WebAssembly responses from the HTTP Archive crawl on 2021-09-01 that matched either `Content-Type` (`application/wasm`) or a file extension

201. <https://webassembly.org/>

(`.wasm`). Then we downloaded all of those<sup>202</sup> with a script<sup>203</sup> that additionally stored the URL, response size, uncompressed size and content hash in a CSV file<sup>204</sup> in the process. We excluded the requests where we repeatedly couldn't get a response due to server errors, as well as those where the content did not in fact look like WebAssembly. For example, some Blazor<sup>205</sup> websites served .NET DLLs<sup>206</sup> with `Content-Type: application/wasm`, even though those are actually DLLs parsed by the framework core, and not WebAssembly modules.

For WebAssembly content analysis, we couldn't use BigQuery directly. Instead, we created a tool<sup>207</sup> that parses all the WebAssembly modules in the given directory and collects numbers of instructions per category, section sizes, numbers of imports/exports and so on, and stores all the stats in a `stats.json` file. After executing it on the directory with downloads from the previous step, the resulting JSON file was imported into BigQuery<sup>208</sup> and joined with the corresponding `summary_requests` and `summary_pages` tables into `httparchive.almanac.wasm_stats` so that each record is self-contained and includes all the necessary information about the WebAssembly request, response and module contents. This final table was then used for all further analysis in this chapter.

Using crawler requests as a source for analysis has its own tradeoffs to be aware of when looking at the numbers in this article:

- First, we didn't have information about requests that can be triggered by user interaction. We included only resources collected during the page load.
- Second, some websites are more popular than others, but we didn't have precise visitor data and didn't take it into account—instead, each detected Wasm usage is treated as equal.
- Finally, in graphs like sizes we counted the same WebAssembly module used across multiple websites as unique usages, instead of comparing only unique files. This is because we are most interested in the global picture of WebAssembly usage across the web pages rather than comparing libraries to each other.

Those tradeoffs are most consistent with analysis done in other chapters, but if you're interested in gathering other statistics, you're welcome to run your own queries against the table `httparchive.almanac.wasm_stats`.

202. <https://github.com/RReverser/wasm-stats/blob/master/downloader/wasms.csv>

203. <https://github.com/RReverser/wasm-stats/blob/master/downloader/index.mjs>

204. <https://github.com/RReverser/wasm-stats/blob/master/downloader/results.csv>

205. <https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor>

206. <https://docs.microsoft.com/en-us/troubleshoot/windows-client/deployment/dynamic-link-library#the-net-framework-assembly>

207. <https://github.com/RReverser/wasm-stats>

208. <https://cloud.google.com/bigquery/docs/batch-loading-data>

## How many modules?

We got 3854 confirmed WebAssembly requests on desktop and 3173 on mobile. Those Wasm modules are used across 2724 domains on desktop and 2300 domains on mobile, which represents 0.06% and 0.04% of all domains on desktop and mobile correspondingly.

Interestingly, when we look at the most popular resulting mime-types, we can see that while `Content-Type: application/wasm` is by far the most popular, it doesn't cover all the Wasm responses—good thing we included other URLs with `.wasm` extension too.



Figure 6.1. Top mime types.

Some of those used `application/octet-stream`—a generic type for arbitrary binary data, some didn't have any `Content-Type` header, and others incorrectly used text types like plain or HTML or even invalid ones like `binary/octet-stream`.

In case of WebAssembly, providing correct `Content-Type` header is important not only for security reasons, but also because it enables a faster streaming compilation and instantiation via `WebAssembly.compileStreaming` and `WebAssembly.instantiateStreaming`.

## How often do we reuse Wasm libraries?

While downloading those responses, we've also deduplicated them by hashing their contents and using that hash as a filename on disk. After that we were left with 656 unique

WebAssembly files on desktop and 534 on mobile.



Figure 6.2. Number of Wasm responses.

The stark difference between the numbers of unique files and total responses already suggests high reuse of WebAssembly libraries across various websites. It's further confirmed if we look at the distribution of cross-origin / same-origin WebAssembly requests:



Figure 6.3. Cross-origin WebAssembly usage.

Let's dive deeper and figure out what those reused libraries are. First, we've tried to deduplicate libraries by content hash alone, but it became quickly apparent that many of those left are still duplicates that differ only by library version.

Then we decided to extract library names from URLs. While it's more problematic in theory due to potential name clashes, it turned out to be a more reliable option for top libraries in practice. We extracted filenames from URLs, removed extensions, minor versions, and suffixes that looked like content hashes, sorted the results by number of repetitions and extracted the top 10 modules for each client. For those left, we did manual lookups to understand which libraries those modules are coming from.

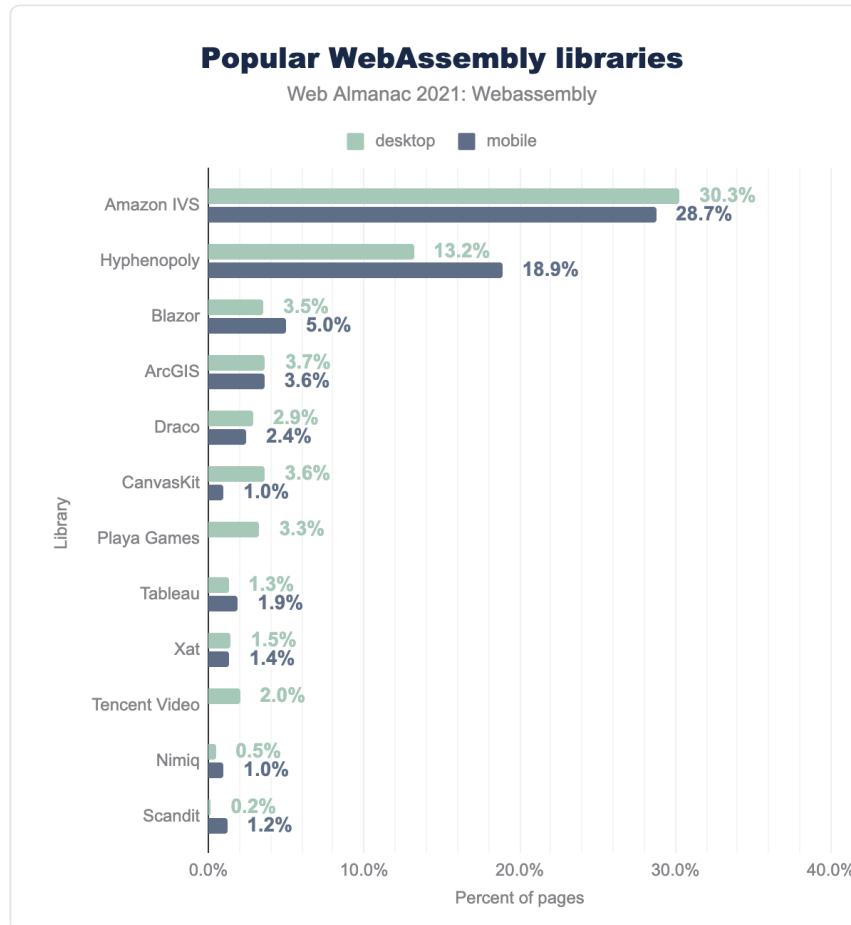


Figure 6.4. Popular WebAssembly libraries.

Almost a third of WebAssembly usages on both desktop and mobile belong to the Amazon Interactive Video Service<sup>209</sup> player library. While it's not open-source, the inspection of the associated JavaScript glue code suggests that it was built with Emscripten<sup>210</sup>.

The next up is Hyphenopoly<sup>211</sup>—a library for hyphenating text in various languages—that accounts for 13% and 19% of Wasm requests on desktop and mobile correspondingly. It's built with JavaScript and AssemblyScript<sup>212</sup>.

209. <https://aws.amazon.com/ivs/>

210. <https://emscripten.org/>

211. <https://github.com/mmattei/Hyphenopoly>

212. <https://www.assemblyscript.org/>

Other libraries from both top 10 desktop and mobile lists account for up to 5% of WebAssembly requests each. Here's a complete list of libraries shown above, with inferred toolchains and links to corresponding homepages with more information:

- Amazon IVS<sup>213</sup> (Emscripten)
- Hyphenopoly<sup>214</sup> (AssemblyScript)
- Blazor<sup>215</sup> (.NET)
- ArcGIS<sup>216</sup> (Emscripten)
- Draco<sup>217</sup> (Emscripten)
- CanvasKit<sup>218</sup> (Emscripten)
- Playa Games<sup>219</sup> (Unity via Emscripten)
- Tableau<sup>220</sup> (Emscripten)
- Xat<sup>221</sup> (Emscripten)
- Tencent Video<sup>222</sup> (Emscripten)
- Nimiq<sup>223</sup> (Emscripten)
- Scandit<sup>224</sup> (Emscripten)

Few more caveats about the methodology and results here:

1. Hyphenopoly loads dictionaries for various languages as tiny WebAssembly files, too, but since those are technically not separate libraries nor are they unique usages of Hyphenopoly itself, we've excluded them from the graph above.
2. WebAssembly file from Playa Games seems to be used by the same game hosted across similarly-looking domains. We count those as individual usages in our query, but, unlike other items in the list, it's not clear if it should be counted as a reusable library.

---

213. <https://aws.amazon.com/ivs/>  
 214. <https://mnater.github.io/Hyphenopoly/>  
 215. <https://dotnet.microsoft.com/apps/aspnet/web-apps/blazor>  
 216. <https://developers.arcgis.com/javascript/latest/>  
 217. <https://google.github.io/draco/>  
 218. <https://skia.org/docs/user/modules/canavaskit/>  
 219. <https://www.playa-games.com/en/>  
 220. [https://help.tableau.com/current/api/js\\_api/en-us/JavaScriptAPI/js\\_api.htm](https://help.tableau.com/current/api/js_api/en-us/JavaScriptAPI/js_api.htm)  
 221. <https://xat.com/>  
 222. <https://intl.cloud.tencent.com/products/vod>  
 223. <https://www.npmjs.com/package/@nimiq/core-web>  
 224. <https://www.scandit.com/developers/>

## How much do we ship?

Languages compiled to WebAssembly usually have their own standard library. Since APIs and value types are so different across languages, they can't reuse the JavaScript built-ins. Instead, they have to compile not only their own code, but also APIs from said standard library and ship it all together to the user in a single binary. What does it mean for the resulting file sizes? Let's take a look:



Figure 6.5. Uncompressed response sizes.

The sizes vary a lot, which indicates a decent coverage of various types of content—from simple helper libraries to full applications compiled to WebAssembly.

We saw sizes of up to 81 MB at the most which may sound pretty concerning, but keep in mind those are uncompressed responses. While they're also important for RAM footprint and start-up performance, one of the benefits of Wasm bytecode is that it's highly compressible, and size over the wire is what matters for download speed and billing reasons.

Let's check sizes of raw response bodies as sent by servers instead:



Figure 6.6. Raw response sizes.

The median is at around 290 KB, meaning that half of usages download below 290 KB, and half are larger. 90% of all Wasm responses stay below 2.6 MB on desktop and 1.4 MB on mobile.

# 44 MB

Figure 6.7. Largest Wasm response downloaded on desktop.

The largest response in the HTTP Archive downloads about 44 MB of Wasm on desktop and 28 MB on mobile.

Even with compression, those numbers are still pretty extreme, considering that many parts of the world still don't have a high-speed internet connection. Aside from reducing the scope of applications and libraries themselves, is there anything websites could do to improve those stats?

## How is Wasm compressed in the wild?

First, let's take a look at compression methods used in these raw responses, based on `Content-Encoding` header. I'll show the mobile dataset here because on mobile bandwidth is even more important, but desktop numbers are pretty similar:

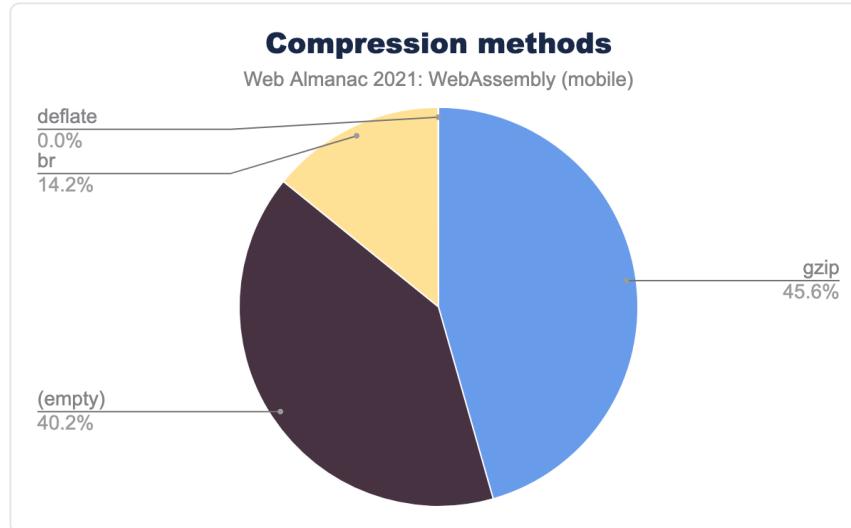


Figure 6.8. Compression methods.

Unfortunately, it shows that ~40% of WebAssembly responses on mobile are shipped without any compression.

# 40.2%

Figure 6.9. Percent of uncompressed WebAssembly responses on mobile.

Another ~46% use gzip, which has been a de-facto standard method on the web for a long time, and still provides a decent compression ratio, but it's not the best algorithm today. Finally, only ~14% use Brotli—a modern compression format that provides an even better ratio and is supported in all modern browsers<sup>225</sup>. In fact, Brotli is supported in every browser that has WebAssembly support too, so there's no reason not to use them together.

## Can we improve compression?

Would it have made a difference? We've decided to recompress all those WebAssembly files with Brotli (compression level 9) to figure it out. The command used on each file was:

<sup>225.</sup> <https://caniuse.com/brotli>

```
brotli -k9f some.wasm -o some.wasm.br
```

Here are the resulting sizes:



*Figure 6.10. Sizes after Brotli compression.*

The median drops from almost 290 KB to almost 240 KB, which is already a pretty good sign. The top 10 percentiles go down from 2.5 MB / 1.4 MB to 2.2 MB / 0.8 MB. We can see significant improvements across all other percentiles, too.

Due to their nature, percentiles don't necessarily fall onto the same files between datasets, so it might be hard to compare numbers directly between graphs and to understand the size savings. Instead, from now on, let's see the savings themselves provided by each optimization, step by step:



Figure 6.11. Brotli response savings.

Median savings are around 40 KB. The top 10% save just under 600 KB on desktop and 330 KB on mobile. The largest savings produced reach as much as 35 MB / 21 MB. Those differences speak in favor of enabling Brotli compression whenever possible, at least for WebAssembly content.

What's also interesting, at the other end of the graph—where we were supposed to see the worst savings—we found regressions of up to 1.4 MB. What happened there? How is it possible that Brotli recompression has made things worse for some modules?

As mentioned above, in this article we've used Brotli with compression level 9, but—and we'll admit, we completely forgot about this until this article—it also has levels 10 and 11. Those levels produce even better results in exchange for a steep performance drop-off, as seen, for example, in Squash benchmarks<sup>26</sup>. Such trade-off makes them worse candidates for the common on-the-fly compression, which is why we didn't use them in this article and went for a more moderate level 9. However, website authors can choose to compress their static resources ahead of time or cache the compression results, and save even more bandwidth without sacrificing CPU time. Cases like these show up as regressions in our analysis, meaning resources can be and, in some cases, already were optimized even better than we did in this article.

<sup>26</sup>. <https://quixdb.github.io/squash-benchmark/#results-table>

## Which sections take up most of the space?

Compression aside, we could also look for optimization opportunities by analyzing the high-level structure of WebAssembly binaries. Which sections are taking up most of the space? To find out, we've summed up section sizes from all the Wasm modules and divided them by the total binary size. Once again, we used numbers from the mobile dataset here, but desktop numbers aren't too far off:



*Figure 6.12. Section size distribution.*

Unsurprisingly, most of the total binary size (~74%) comes from the compiled code itself, followed by ~19% for embedded static data. Function types, import/export descriptors and such comprise a negligible part of the total size. However, one section type stands out—it's custom sections, which account for ~6.5% of total size in the mobile dataset.

# 6.5%

*Figure 6.13. Portion of custom sections in the total binary size of mobile dataset.*

Custom sections are mainly used in WebAssembly for 3rd-party tooling—they might contain information for type binding systems, linkers, DevTools and such. While all of those are legitimate use-cases, they are rarely necessary in production code, so such a large percentage is suspicious. Let's take a look at what they are in top 10 files with largest custom sections:

<b>URL</b>	<b>Size of Custom Sections</b>	<b>Custom Sections</b>
.../dotnet.wasm <sup>227</sup>	15,053,733	<code>name</code>
.../unity.wasm.br?v=1.0.8874 <sup>228</sup>	9,705,643	<code>name</code>
.../nanoleq-HTML5-Shipping.wasmgz <sup>229</sup>	8,531,376	<code>name</code>
.../export.wasm <sup>230</sup>	7,306,371	<code>name</code>
.../c0c43115a4de5de0.../northstar_api.wasm <sup>231</sup>	6,470,360	<code>name,</code> <code>external_debug_info</code>
.../9982942a9e080158.../northstar_api.wasm <sup>232</sup>	6,435,469	<code>name,</code> <code>external_debug_info</code>
.../ReactGodot.wasm <sup>233</sup>	4,672,588	<code>name</code>
.../v18.0-591dd9336/trace_processor.wasm <sup>234</sup>	2,079,991	<code>name</code>
.../v18.0-615704773/trace_processor.wasm <sup>235</sup>	2,079,991	<code>name</code>
.../canvaskit.wasm <sup>236</sup>	1,491,602	<code>name</code>

Figure 6.14. Largest custom sections.

All of those are almost exclusively the `name` section which contains function names for basic debugging. In fact, if we keep looking through the dataset, we can see that almost all of those custom sections contain just the debug information.

## How much can we save by stripping debug info?

While debug information is useful for local development, those sections can be hefty—they take over 14 MB before compression in the table above. If you want to be able to debug production issues users are experiencing, a better approach might be to strip the debug information out of the binary using `llvm-strip`, `wasm-strip` or `wasm-opt --strip-debug` before shipping, collect raw stacktraces and match them back to source locations locally, using the

227. [https://gallery.platform.uno/package\\_85a43e09d7152711f12894936a8986e20694304a/dotnet.wasm](https://gallery.platform.uno/package_85a43e09d7152711f12894936a8986e20694304a/dotnet.wasm)  
 228. <https://cdn.decentraland.org/dcl/unity-renderer/1.0.12536-20210902152600.commit-86fe4be/unity.wasm.br?v=1.0.8874>  
 229. <https://nanoleq.com/nanoleq-HTML5-Shipping.wasmgz>  
 230. <https://convertmodel.com/export.wasm>  
 231. [https://webaset.akm.imvu.com/asset/c0c43115a4de5de0/build/northstar/js/northstar\\_api.wasm](https://webaset.akm.imvu.com/asset/c0c43115a4de5de0/build/northstar/js/northstar_api.wasm)  
 232. [https://webaset.akm.imvu.com/asset/9982942a9e080158/build/northstar/js/northstar\\_api.wasm](https://webaset.akm.imvu.com/asset/9982942a9e080158/build/northstar/js/northstar_api.wasm)  
 233. <https://supertcf.com/ReactGodot.wasm>  
 234. [https://ui.perfetto.dev/v18.0-591dd9336/trace\\_processor.wasm](https://ui.perfetto.dev/v18.0-591dd9336/trace_processor.wasm)  
 235. [https://ui.perfetto.dev/v18.0-615704773/trace\\_processor.wasm](https://ui.perfetto.dev/v18.0-615704773/trace_processor.wasm)  
 236. <https://unpkg.com/canvaskit-wasm@0.25.1/bin/profiling/canvaskit.wasm>

original binary.

It would be interesting to see how much much stripping this debug information would save us in combination with Brotli, vs. just Brotli from the previous step. However, most modules in the dataset don't have custom sections so any percentiles below 90 would be useless:



Figure 6.15. *strip-debug + Brotli savings*.

Instead, let's take a look at the distribution of savings only over files that do have custom sections:



Figure 6.16. *strip-debug + Brotli savings*.

As can be seen from the graph, some file's custom sections are negligibly small, but the median is at 54 KB and the 90 percentile is at 247 KB on desktop and 118 KB on mobile. The largest savings we could get were at 2.4 MB / 1.3 MB for the largest Wasm binaries on desktop and mobile, which is a pretty noticeable improvement, especially on slow connections.

You might have noticed that the difference is a lot smaller than raw sizes of custom sections from the table above. The reason is that the `name` section, as its name suggests, consists mostly of function names, which are ASCII strings with lots of repetitions, and, as such, are highly compressible.

There are a few outliers where the process of removing custom sections with `llvm-strip` made some changes to the WebAssembly module that made it smaller before compression, but slightly larger after the compression. Such cases are rare though, and the difference in size is insignificant compared to the total size of the compressed module.

## How much can we save via `wasm-opt`?

`wasm-opt` from the Binaryen<sup>237</sup> suite is a powerful optimization tool that can improve both size and performance of the resulting binaries. It's used in major WebAssembly toolchains such as Emscripten, wasm-pack and AssemblyScript to optimize binaries produced by the underlying compiler.

237. <https://github.com/WebAssembly/binaryen>

It provides significant size savings on both uncompressed and compressed real-world benchmarks:



Figure 6.17. `wasm-opt` uncompressed size benchmarks.



Figure 6.18. `wasm-opt` + Brotli size benchmarks.

We've decided to check the performance of `wasm-opt` on the collected HTTP Archive dataset as well, but there's a catch.

As mentioned above, `wasm-opt` is already used by most compiler toolchains, so most of the modules in the dataset are already its resulting artifacts. Unlike in compression analysis above, there's no way for us to reverse existing optimizations and run `wasm-opt` on the originals. Instead, we're re-running `wasm-opt` on pre-optimized binaries, which skews the results. This is the command we've used on binaries produced after the strip-debug step:

```
wasm-opt -O all some.wasm -o some.opt.wasm
```

Then, we compressed the results to Brotli and compared to the previous step, as usual.

While the resulting data is not representative of real-world usage and not relevant to regular consumers who should use `wasm-opt` as they normally do, it might be useful to consumers like CDNs that want to run optimizations at scale, as well as to the Binaryen team itself:

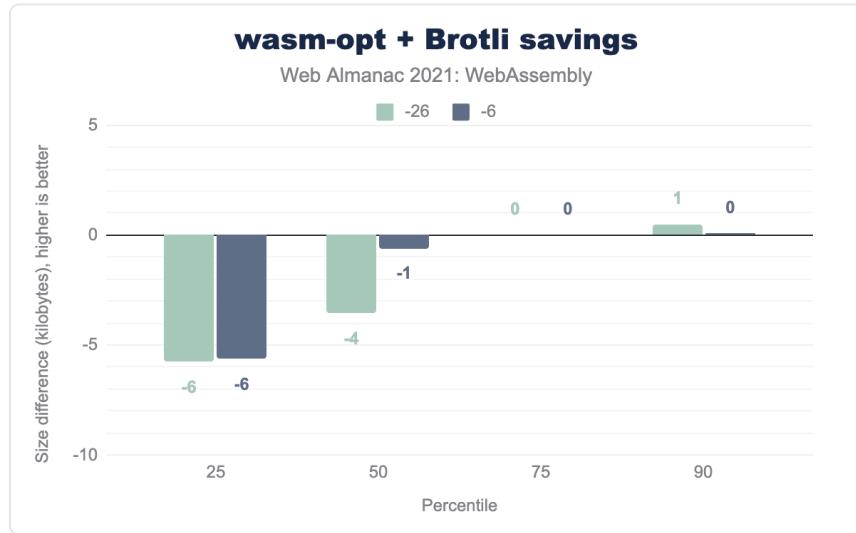


Figure 6.19. `wasm-opt` + Brotli savings.

The results in the graph are mixed, but all changes are relatively small, up to 26 KB. If we included outliers (0 and 100 percentiles), we'd see more significant improvements of up to 1 MB on desktop and 240 KB on mobile on the best end, and regressions of 255 KB on desktop and 175 KB on mobile on the worst end.

The significant savings in a small percentage of files mean they were likely not optimized before publishing on the web. But why are the other results so mixed?

If we look at the uncompressed savings, it becomes more clear that, even on our dataset, `wasm-opt` consistently keeps files either roughly the same size or still improves size slightly further in majority of cases, and produces significant savings for the unoptimized files.



Figure 6.20. Uncompressed `wasm-opt` savings.

This suggests several reasons for the surprising distribution in the post-compression graph:

1. As mentioned above, our dataset does not resemble real-world `wasm-opt` usage as the majority of the files have been already pre-optimized by `wasm-opt`. Further instruction reordering that improves uncompressed size a bit further, is bound to make certain patterns either more or less compressible than others, which, in turn, produces statistical noise.
2. We use default `wasm-opt` parameters, whereas some users might have tweaked `wasm-opt` flags in a way that produces even better savings for their particular modules.
3. As mentioned earlier, the network (compressed) size is not everything. Smaller WebAssembly binaries tend to mean faster compilation in the VM, less memory consumption while compiling, and less memory to hold the compiled code. `wasm-opt` has to strike a balance here, which might also mean that the compressed size might sometimes regress in favor of better raw sizes.
4. Finally, some of the regressions look like potentially valuable examples to study and improve that balance. We've reported them back<sup>238</sup> to the Binaryen team so that they could look deeper into potential optimizations.

<sup>238</sup>. <https://github.com/WebAssembly/binaryen/issues/4322>

## What are the most popular instructions?

We've already glimpsed at the contents of Wasm when sliced by section kinds above. Let's take a deeper look at the contents of the code section—the largest and the most important part of a WebAssembly module.

We've split instructions into various categories and counted them across all the modules together:



Figure 6.21. Instruction kinds.

One surprising takeaway from this distribution is that local var operations—that is, `local.get`, `local.set` and `local.tee`—comprise the largest category—36%, far ahead from the next few categories—inline constants (15.2%), load/store operations (14.7%) and all the math and logical operations (14.3%). Local var operations are usually generated by compilers as a result of optimization passes in compilers. They downgrade expensive memory access operations to local variables where possible, so that engines can subsequently put those local variables into CPU registers, which makes them much cheaper to access.

It's not actionable information for developers compiling to Wasm, but something that might be interesting to engine and tooling developers as a potential area for further size optimizations.

## What's the usage of post-MVP extensions?

Another interesting metric to look at is post-MVP Wasm extensions. While WebAssembly 1.0 was released several years ago, it's still actively developed and grows with new features over time. Some of those improve code size by moving common operations to the engines, some provide more powerful performance primitives, and others improve developer experience and integration with the web. On the official feature roadmap<sup>239</sup> we track support for those proposals across latest versions of every popular engine.

Let's take a look at their adoption in the Almanac dataset too:

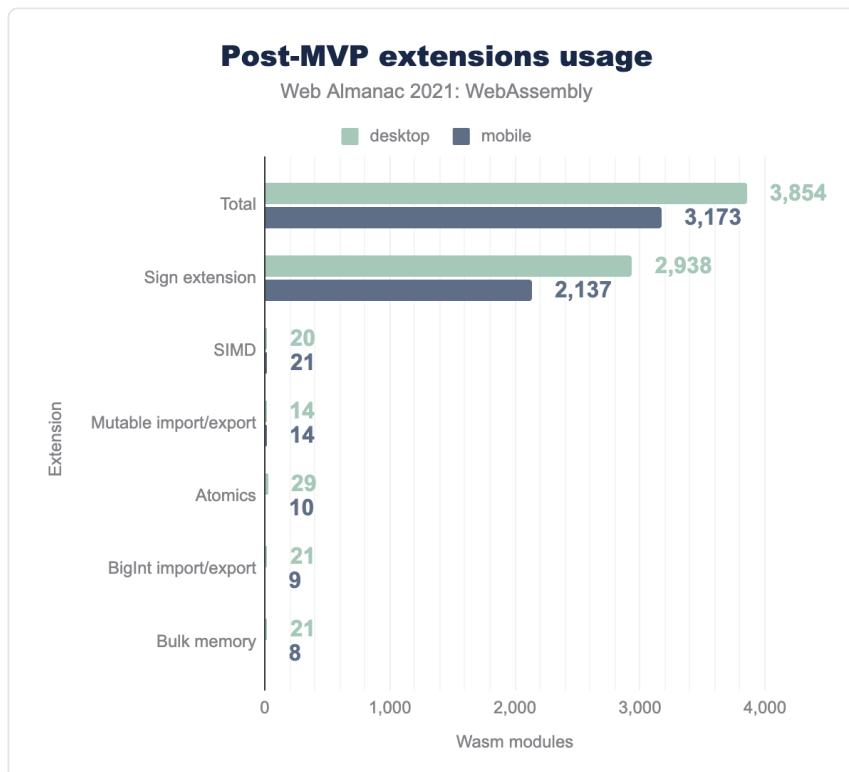


Figure 6.22. Post-MVP extensions usage.

One feature stands out—it's the sign-extension operators proposal<sup>240</sup>. It was shipped in all browsers not too long after the MVP, and enabled in LLVM (a compiler backend used by Clang /

239. <https://webassembly.org/roadmap/>

240. <https://github.com/WebAssembly/sign-extension-ops/blob/master/proposals/sign-extension-ops/Overview.md>

Emscripten and Rust) by default, which explains its high adoption rate. All other features currently have to be enabled explicitly by the developer at compilation time.

For example, non-trapping float-to-int conversions<sup>241</sup> is very similar in spirit to sign-extension operators—it also provides built-in conversions for numeric types to save some code size—but it became uniformly supported only recently with the release of Safari 15. That's why this feature is not yet enabled by default, and most developers don't want the complexity of building and shipping different versions of their WebAssembly module to different browsers without a very compelling reason. As a result, none of the Wasm modules in the dataset used those conversions.

Other features with zero detected usages—multi-value, reference types and tail calls—are in a similar situation: they could also benefit most WebAssembly use-cases, but they suffer from incomplete compiler and/or engine support.

Among the remaining, used, features, two that are particularly interesting are SIMD and atomics. Both provide instructions for parallelizing and speeding up execution at different levels: SIMD<sup>242</sup> allows to perform math operations on several values at once, and atomics provide a basis for multithreading in Wasm<sup>243</sup>. Those features are not enabled by default, require specific use-cases, and multithreading in particular requires using special APIs in the source code as well as additional configuration to make the website cross-origin isolated<sup>244</sup> before it can be used on the web. As a result, a relatively low usage level is unsurprising, although we expect them to grow over time.

## Conclusion

While WebAssembly is a relatively new and somewhat niche participant on the web, it's great to see its adoption across a variety of websites and use-cases, from simple libraries to large applications.

In fact, we could see that it integrates so well into the web ecosystem, that many website owners might not even know they already use WebAssembly—to them it looks like any other 3rd-party JavaScript dependency.

We found some room for improvement in shipped sizes which, through further analysis, appears to be achievable via changes to compiler or server configuration. We've also found some interesting stats and examples that might help engine, tooling and CDN developers to understand and optimize WebAssembly usage at scale.

---

<sup>241</sup> <https://github.com/WebAssembly/nontrapping-float-to-int-conversions/blob/master/proposals/nontrapping-float-to-int-conversion/Overview.md>

<sup>242</sup> <https://v8.dev/features/simd>

<sup>243</sup> <https://web.dev/webassembly-threads/>

<sup>244</sup> <https://web.dev/coop-coep/>

We'll be tracking those stats over time and return with updates in the next edition of the Web Almanac.

## Author

---



Ingvar Stepanyan

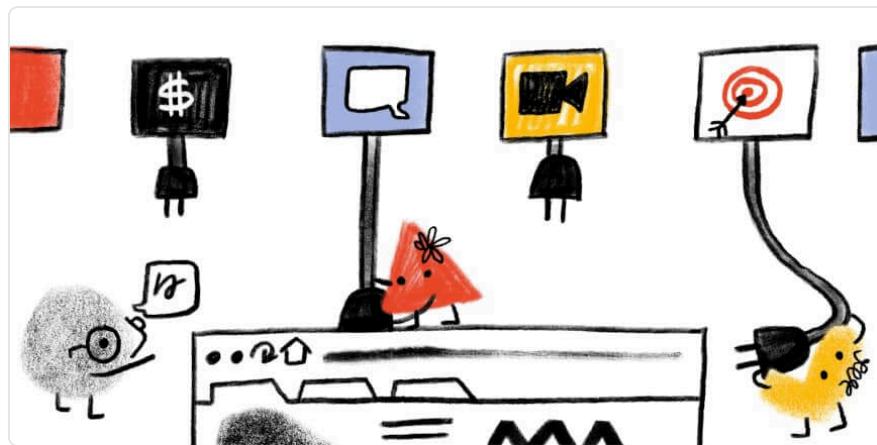
Twitter: [@RReverser](https://twitter.com/RReverser) GitHub: [RReverser](https://github.com/RReverser) Website: <https://rreverser.com/>

Ingvar is a passionate D2D (developer-to-developer) programmer who's always working on improving developer experience through better tools, specs and documentation. He currently works as a WebAssembly Developer Advocate on the Google Chrome team.

---

# Part I Chapter 7

# Third Parties



**Written by Barry Pollard**

Reviewed by Patrick Hulce, Andy Davies, Simon Hearne, and Harry Roberts

Analyzed by Barry Pollard

Edited by Rick Viscomi

## Introduction

Ah third parties, the solution to so many problems on the web... and cause of so many others! Fundamentally, the web has always been about interconnectivity and sharing. Using third-party content on a website is a natural extension of that and was first set into motion with the introduction of the `<img>` element in HTML 2.0; we have been able to hyperlink external content straight into our documents ever since. This has only grown with the introduction of CSS, and JavaScript allowing part (or all!) of the page to be changed completely just by including a seemingly simple `<link>` or `<script>` element.

Third parties provide a never-ending collection of images, videos, fonts, tools, libraries, widgets, trackers, ads, and anything else you can imagine embedding into our web pages. This enables even the most non-technical to be able to create and publish content to the web. Without third parties, the web would likely be a very boring, text-based, academic medium instead of the rich, immersive, complex platform that is so integral to the lives of many of us today.

However, there is a dark side to using third-party content on the web. An innocuous inclusion of an image or a helpful library opens the floodgates to all sorts of performance, privacy, and security implications that many developers do not consider fully. Speak to any professionals in those industries and they will lament the use of third-party content making their lives more difficult. Scrutiny is surely only going to grow with performance getting extra attention through the Core Web Vitals initiative from Google<sup>245</sup>, increased focus on privacy from governments and individuals, and the ever-increasing threat of exploitable vulnerabilities and malicious threats inherent to the web.

In this chapter we're going to have a look at the state of third parties on the web: how much are we using them, what are we using them for, and has our usage changed over the last year, particularly given the three concerns listed above? These are questions I'm looking to answer here.

## Definitions

We may have different ideas of what constitutes a “third party” or “using third-party content”, so we'll start with a definition of what we consider a third party to be for this chapter:

### “Third party”

We use the same definition of third party as we have in the 2019<sup>246</sup> and 2020<sup>247</sup> editions, though a slightly different interpretation of it will exclude one category this year, as we'll discuss in the next section.

A third party is an entity outside the primary site-user relationship, i.e. 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 an example of 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

245. <https://web.dev/vitals/>

246. <https://almanac.httparchive.org/en/2019/third-parties>

of a third-party resource is one 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 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—assuming it passes the “more than 50 pages criteria”, which it may well do based on domain, even if the resource itself is unique to that website. For example, first-party images served over a CDN on a third-party domain are considered *third-party content*.

## Third-party categories

This year we will, again, be drawing heavily on the third-party-web<sup>248</sup> repository from Patrick Hulce<sup>249</sup> to help us identify and categorize third parties. This repository categorizes commonly used third-party URLs into the following categories:

- **Ad** - These scripts are part of advertising networks, either serving or measuring.
- **Analytics** -These scripts measure or track users and their actions. There's a wide range in impact here depending on what's being tracked.
- **CDN** - These are a mixture of publicly hosted open source libraries (e.g. jQuery) served over different public CDNs and private CDN usage.
- **Content** - These scripts are from content providers or publishing-specific affiliate tracking.
- **Customer Success** - These scripts are from customer support/marketing providers that offer chat and contact solutions. These scripts are generally heavier in weight.
- **Hosting** - These scripts are from web hosting platforms (WordPress, Wix, Squarespace, etc.).
- **Marketing** - These scripts are from marketing tools that add popups/newsletters/etc.
- **Social** - These scripts enable social features.
- **Tag Manager** - These scripts tend to load lots of other scripts and initiate many tasks.

248. <https://github.com/patrickhulce/third-party-web/blob/master/data/entities.js>

249. <https://twitter.com/patrickhulce>

- **Utility** - These scripts are developer utilities (API clients, site monitoring, fraud detection, etc.).
- **Video** - These scripts enable video player and streaming functionality.
- **Other** - These are miscellaneous scripts delivered via a shared origin with no precise category or attribution.

*Note: 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. For example, putting Cloudflare in front of a page would not influence its first-party designation according to our criteria.*

One change that we have made to our methodology this year is to remove the Hosting category from our analysis. If you happen to use WordPress.com for your blog, or Shopify for your ecommerce platform, then we're going to ignore other requests for those domains by that site as not truly "third-party", as they are in many ways part of hosting on those platforms. Similar to the note above, we do not consider CDNs in front of a page to be "third party". In reality this made very little difference to the numbers, but we feel it's a more accurate reflection of what we should consider "third party" by the above definition, and also aligns more closely with how the other chapters use this term.

## 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<sup>250</sup>, or other similar legislation.
- Only the home pages are tested. Other pages may have different third-party requirements.
- Some of the lesser-used third-party domains are grouped into the *unknown* category. As part of this analysis, we submitted more categories for the top-used domains to improve the third-party-web dataset.

Learn more about our methodology.

250. [https://en.wikipedia.org/wiki/General\\_Data\\_Protection\\_Regulation](https://en.wikipedia.org/wiki/General_Data_Protection_Regulation)

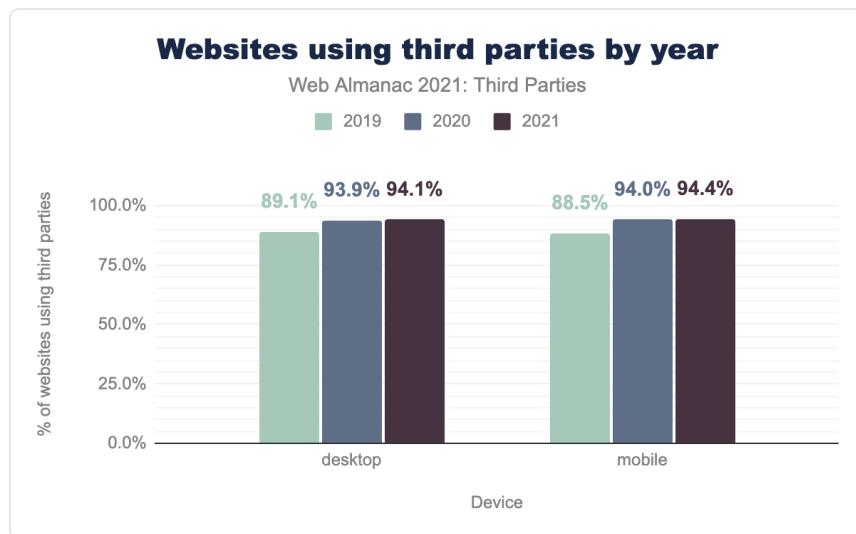
## Prevalence

So how much are third parties used? Well, the answer is a lot!

# 94.4%

*Figure 7.1. Percentage of mobile sites using at least one third-party resource.*

A staggering 94.4% of mobile sites and 94.1% of desktop sites use at least one third-party resource. Even with our newer restrictive definition of third parties, this represents a continued growth from when the Web Almanac started in 2019<sup>251</sup>.



*Figure 7.2. Websites using third parties by year.*

Rerunning the last three annual Web Almanac datasets with the new, stricter definition, we see in the chart above that our usage of third parties on our website has grown slightly on last year by 0.2% on desktop and 0.4% on mobile.

<sup>251</sup> <https://almanac.httparchive.org/en/2019/third-parties>

# 45.9%

*Figure 7.3. Percentage of requests which are third-party.*

45.9% of requests on mobile and 45.1% of requests on desktop are third-party requests, which is similar to last year's results<sup>252</sup>.

It would appear that privacy-preserving regulations like GDPR<sup>253</sup> and CCPA<sup>254</sup> are not dampening our appetite for third-party usage. Though it should be remembered that our methodology is to test websites from US data centers and so may be served different content because of that.

So, we know nearly all sites use third parties, but how many do they use?



*Figure 7.4. Number of third parties per website.*

Looking at the spread, we see there is a large variance with websites only using two third parties—measured as the number of distinct third-party hostnames—at the 10th percentile, up to 89 or 91 at the 90th percentile.

Note that the 90th percentile is down a bit from last year's analysis<sup>255</sup>, where we had 104 and 106 third parties for desktop and mobile respectively, but this looks to be due to restricting our

252. <https://almanac.httparchive.org/en/2020/third-parties>

253. [https://en.wikipedia.org/wiki/General\\_Data\\_Protection\\_Regulation](https://en.wikipedia.org/wiki/General_Data_Protection_Regulation)

254. [https://en.wikipedia.org/wiki/California\\_Consumer\\_Privacy\\_Act](https://en.wikipedia.org/wiki/California_Consumer_Privacy_Act)

255. <https://almanac.httparchive.org/en/2020/third-parties#fig-2>

domains to assets used by 50 websites or more this year, which was not done for this statistic last year.

The median website uses 21 third parties on mobile and 23 on desktop, which still seems like quite a lot!

## Third party prevalence by rank



Figure 7.5. Websites using third parties by rank.

This year we have access to the Chrome UX Report (CrUX) “rank”<sup>256</sup> for each website. This is a popularity assignment for each site, which allows us to group our data into the top 1,000 most-used sites (based on page views), top 10,000 most-used sites, etc. Slicing the data by this popularity rank shows that there is a slight decrease in third-party usage for the less popular websites, but it never dips below 93.3%, again reiterating that pretty much all websites love to include at least one third party.

However, what does change is the number of third parties used by website:

<sup>256</sup>. <https://developers.google.com/web/updates/2021/03/crux-rank-magnitude>

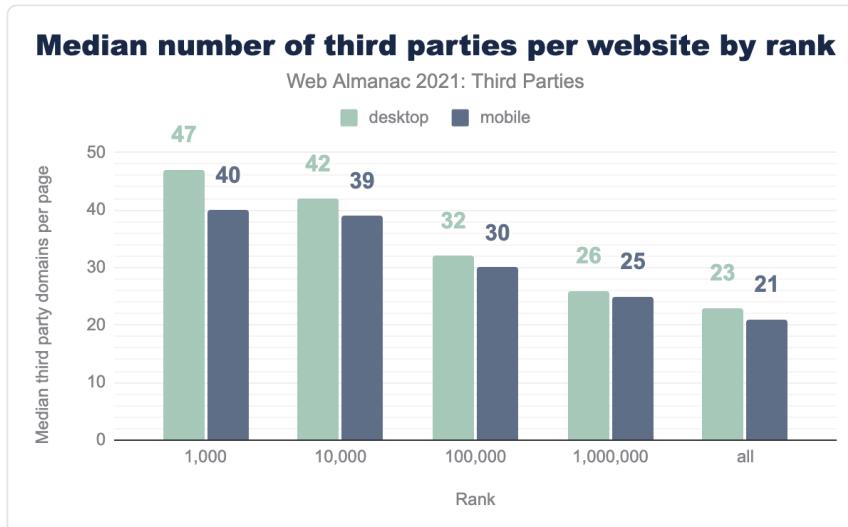


Figure 7.6. Median number of third parties per website by rank.

Looking at the median (50th percentile) statistics, we see a marked decline as we go up the rankings, with the most popular websites using twice as many third parties as the whole dataset. We'll see in a moment that that is driven almost entirely by ads. It is perhaps unsurprising that these are much more prevalent on more popular websites, with more eyeballs to monetize.

## Third-party type

Our analysis shows we're using third parties a lot, but what are we using them for? Looking at the categories of each third-party request, we see the following:

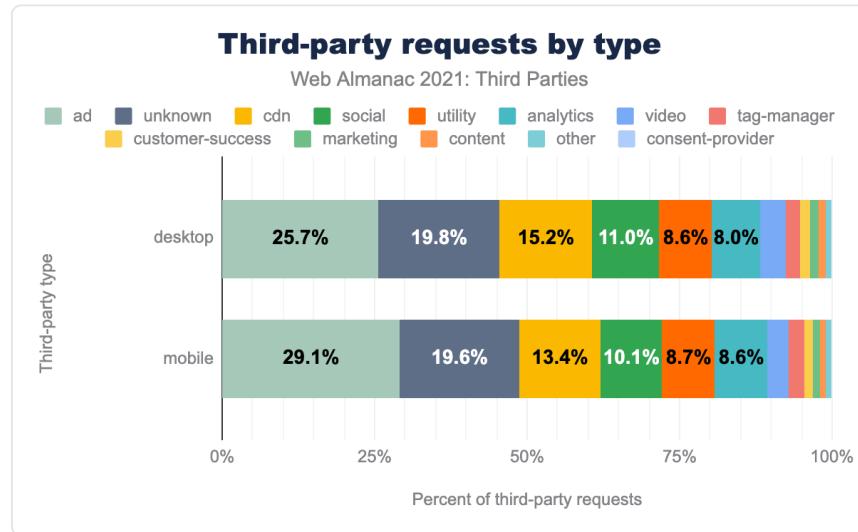


Figure 7.7. Third-party requests by type.

Ads are the most common third-party requests, followed by “unknown”—a collection of various uncategorized or lesser-used sites—then CDN, social, utility, and analytics. So, while some categories are more popular than others, what’s perhaps the bigger takeaway here is how varied third-party usage is. They really are used for all sorts of reasons, rather than one or two use cases dominating all the others.

## Third-party requests by type and rank



Figure 7.8. Median third-party requests by type and rank.

Splitting the requests by rank and category, we see the reason for the larger number of requests discussed previously: ads are much more heavily used on the more popular sites.

Note this chart shows the median number of requests for each category, by rank, but not every category is used on every page, explaining why the totals per rank are much higher than the median number of requests per rank from the previous chart.

## Content types

Taking an alternative view on the data, let's see what type of content we're getting back from all those third-party requests.



Figure 7.9. Third-party usage by content type.

Unsurprisingly, JavaScript, images, and HTML comprise the majority of third-party requests. JavaScript is used by most third parties to add functionality, whether that be in ads, trackers, or libraries. Similarly, the high usage of images is to be expected, as they will include the 1-pixel blank images so beloved of tracking solutions.

The high usage of HTML may seem surprising initially (surely documents would be the prevalent form of HTML and they would be first-party requests?), but our investigation showed them mostly to be iframes, which makes much more sense as they are often used to house ads, or other widgets (e.g. YouTube serves an HTML document in an iframe including the player, rather than just the video itself).

So based purely on the number of requests, third parties seem to be adding functionality more so than content—though that's a little misleading since, as per the YouTube example, some third parties add functionality in order to enable the content.

## Third-party requests by content type and category

Web Almanac 2021: Third Parties (mobile)



Figure 7.10. Third-party requests by content type and category (mobile).

Splitting the requested content types by the type of third party, we see the prevalence of those three main types (scripts, images, and HTML) across most types, though the worrying amount of JavaScript (even for video type!) is already apparent. The above chart is for mobile, but the desktop picture is very similar.

## Third-party bytes by content type and category

Web Almanac 2021: Third Parties (mobile)



Figure 7.11. Third-party requests by content type and category.

When looking by bytes, rather than by requests, the amount of JavaScript is even more worrying. Again, we've shown mobile here, but there are no major differences for desktop.

To quote Addy Osmani<sup>257</sup> (twice in the same sentence!) from his “Cost of JavaScript”<sup>258</sup> post, “byte-for-byte, JavaScript is still the most expensive resource we send”, and “a 200 KB script and a 200 KB image have very different costs”. Some categories like Analytics, Consent Provider, and Tag Manager are pretty much all JavaScript, while others like Ad and Customer Success are not far behind. We'll return to the performance impact of using third-party resources, which is often caused by costly use of JavaScript.

## Third-party domains

Who are we loading all these third-party requests from? Most of these names won't be surprising, but the prevalence of one name just reiterates the dominance that company has across a number of different categories:

<sup>257</sup>. <https://twitter.com/addyosmani>  
<sup>258.</sup> <https://medium.com/@addyosmani/the-cost-of-javascript-in-2018-7d8950fbb5d4>



Figure 7.12. Top 15 third parties by usage.

Google takes 8 of the top 15 most-used third parties—including the top 6 spots!—and no else comes close. Google is a market leader in Analytics, Fonts, Ads, Accounts, Tag Managers, and Video to name but a few. A staggering 62.7% of mobile websites use Google Analytics, and almost as many use Google Fonts, with Ads, Accounts and Tag Manager usage not far behind in the 42%-49% range.

The first non-Google entity is Facebook, with comparatively low usage of 29.2%. This is followed by Cloudflare's CDN fronting popular libraries and other resources. Despite being listed as amp.cloudflare.com, it also includes the much larger cdnjs.cloudflare.com—this has been updated to show the more commonly used domain for next year.

After this we're back to Google with YouTube, and Maps two spots later. The remaining spots are filled with CDNs for other popular libraries and tools.

## Performance impact of third parties

Using third parties can have a noticeable impact on performance. That's not necessarily a

consequence of them being a third party per se. The same functionality implemented by a site owner as a first-party resource can often be less performant, given the expertise the third party should have on the particular field.

So, performance isn't necessarily impacted by the fact that the resources are third-party, it's more of a matter of what those resources are doing. And most third-party usage depends on the third-party service, rather than just as a place to serve it from.

However, a third party's business is in allowing their content or service to be hosted on many websites. Third parties have a duty to ensure that they minimize the negative impact of that dependency. This is an especially important duty given that site owners often have limited control over and influence on the performance impact of third parties other than to use them or not.

## Using third-party domains versus self-hosting

There is a definite cost to connecting to another domain, even though most third parties will be using globally distributed, high-performance CDNs, and many web performance advocates (including this author!) recommend self-hosting where possible to avoid this penalty. This is particularly relevant now that all the major browsers have moved away from sharing caches between origins, so the claim that once one site has downloaded that resource, other sites visited can also benefit from it is no longer true. Though this was a questionable claim even in the past, given the number of versions of libraries, and limitations of the HTTP cache.

Saying that, rarely is life as definitive as we would like and, in some cases self-hosting may actually cost performance. This author has written before how the question on whether to self-host Google Fonts<sup>259</sup> is not as clear cut as it might seem and requires a degree of expertise to ensure you are replicating all that Google Fonts does for you in the performance front. To avoid that hassle you can just use the hosted version, and ensure you're reducing the performance impact as much as possible, as discussed by Harry Roberts<sup>260</sup> in his The Fastest Google Fonts<sup>261</sup> post.

Similarly, image CDNs can optimize media better than most first-parties and, more importantly, can do this automatically without the need for manual steps that will inevitably be skipped or done incorrectly on occasion.

---

259. <https://www.tunetheweb.com/blog/should-you-self-host-google-fonts/>

260. <https://twitter.com/csswizardry>

261. <https://csswizardry.com/2020/05/the-fastest-google-fonts/>

## Popular third parties embeds and their performance impact

To try to understand the performance impact of third parties, we will look at some of the most popular third-party embeds. Some of these have gotten a bad name in web performance circles, so let's see if the bad reputation is really deserved. To do that, we'll be making use of two Lighthouse audits: Eliminate render blocking resources<sup>262</sup> and Reduce the impact of third-party code<sup>263</sup>, based on some similar research<sup>264</sup> by Houssein Djirdeh<sup>265</sup>.

### Popular third parties and their impact on render

To understand third parties' impact on rendering, we've analyzed how sites resources perform on Lighthouse's render-blocking resources audit, and identified which are third-parties by cross-referencing them with the third-party-web dataset.

---

262. <https://web.dev/render-blocking-resources/>

263. <https://web.dev/third-party-summary/>

264. [https://docs.google.com/spreadsheets/d/1Td-4qFJuBzxp8af\\_if5iBC0Lkqm\\_OROb7\\_2OcbxrU\\_g/edit?usp=sharing&resourcekey=0-ZCfve5cngWxFO-sv5pLRzg](https://docs.google.com/spreadsheets/d/1Td-4qFJuBzxp8af_if5iBC0Lkqm_OROb7_2OcbxrU_g/edit?usp=sharing&resourcekey=0-ZCfve5cngWxFO-sv5pLRzg)

265. <https://twitter.com/hdjirdeh>

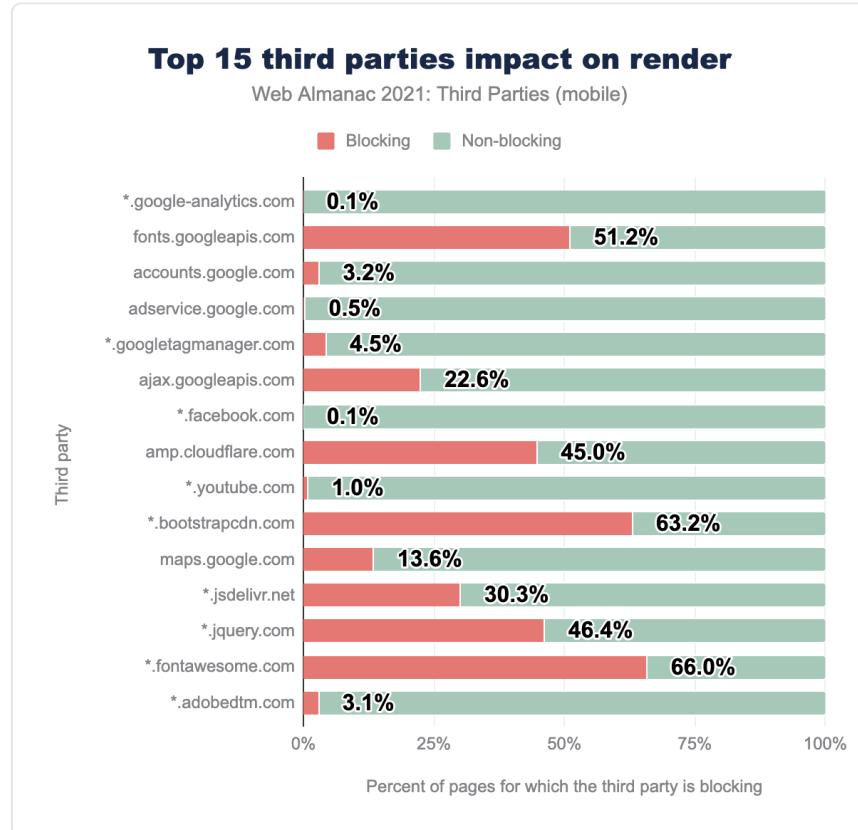


Figure 7.13. Top 15 third parties impact on render.

The top 15 most popular third parties are shown above along with the percentage of resources they block on the initial render of the page.

On the whole this is a positive story; most do not block rendering, and those that do are for common libraries associated with layout (e.g. bootstrap) or fonts that perhaps should block initial render (this author doesn't agree that using `font-display: swap` or `optional` is a good thing).

Often third-party embeds advise using `async` or `defer` to avoid blocking rendering, and it looks like this might be the case for many of them.

## Popular third parties and their impact on main thread

Lighthouse has a Reduce the impact of third-party code<sup>266</sup> audit that lists the main-thread times of all third-party resources. So how long do the most popular ones block the main thread for?



Figure 7.14. Main-thread blocking time of top 15 third parties.

Here we see YouTube sticking out like a sore thumb so let's delve into that a little more:

266. <https://web.dev/third-party-summary/>

## YouTube



Figure 7.15. YouTube's impact on the main thread.

We can see a huge impact of 1.6 seconds of main-thread activity at the median (50th percentile), rising to a shocking 4.6 seconds of main-thread blocking at the 90th percentile (still meaning 10% of websites have a worse impact than even that!). It should be remembered however that these are throttled, lab-simulated timings, so many real users may not be experiencing this level of impact, but it is still a lot.

It's also apparent that the impact increases with transfer size—perhaps not surprising as there is more to process. And remember that our crawl does not interact with these videos, so these are either auto-playing videos, or the YouTube player itself causing all this use.

Let's dig a little deeper into some of the other third party embeds on our list.

## Google Analytics



Figure 7.16. Google Analytics' impact on the main thread.

Google Analytics is pretty good, so obviously a lot of work has gone into optimizing this, given all that it tracks.

## Google/Doubleclick Ads

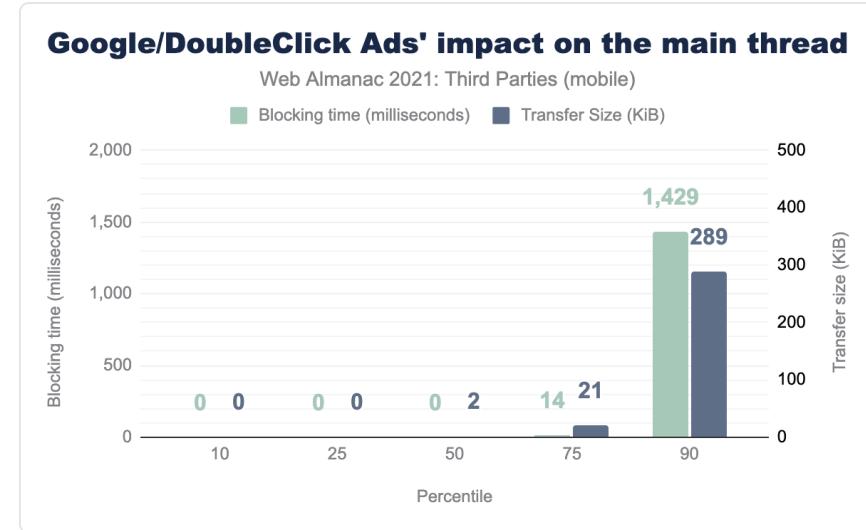


Figure 7.17. Google/DoubleClick Ads' impact on the main thread.

Google Ads was doing so well, until we hit the 90th percentile, when it got blown off the chart. Again, a reminder that this means 10% of websites have worse numbers than these.

## Google Tag Manager



Figure 7.18. Google Tag Manager's impact on the main thread.

Google Tag Manager fares much better than expected to be honest. This author has seen some horrific GTM implementations, overloaded with old tags and triggers that are no longer used. But GTM seems to do well at not blocking the main thread for too long in our test page loads.

## Facebook



Figure 7.19. Facebook's impact on the main thread.

Facebook also isn't as resource intensive as I thought it would be. Facebook embeds of posts seem to be less popular than Twitter embeds, so these will likely be Facebook retargeting trackers. These trackers should be working silently in the background and not impacting the main thread at all, so it's apparent there is still more work for Facebook to do here. I've even had good success in not using the Facebook JavaScript API and using pixel tracking through Google Tag Manager<sup>267</sup> without losing any functionality, and would encourage others to consider this option.

267. <https://www.tunetheweb.com/blog/adding-controls-to-google-tag-manager/#pixels>

## Google Maps



Figure 7.20. Google Maps' impact on the main thread.

Google Maps definitely needs some improvement. Especially as it's often present as a small extra piece on a page, rather than the main content. As a website owner, this highlights the importance of only including the Google Maps code on pages that require it.

## Twitter

And finally, let's look at one further down the list: Twitter.

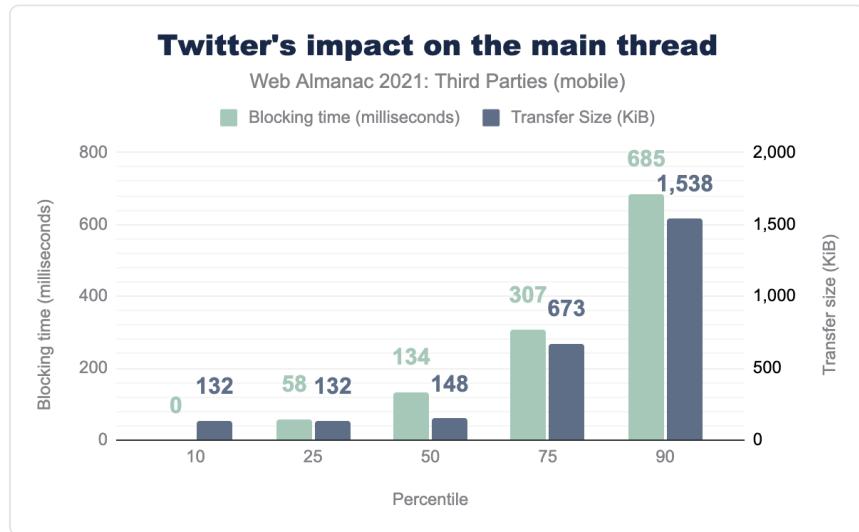


Figure 7.21. Twitter's impact on the main thread.

Twitter as a third-party can be used in one of two ways: as a retargeting advertising tracker, and as a way of embedding tweets. Embedding tweets in pages is more popular than other social networks. However it has been called out as having an undue impact on the page by many in the web performance community, including Matt Hobbs<sup>268</sup> in his Using Puppeteer and Squoosh to fix the web performance of embedded tweets<sup>269</sup> post. Our analysis backs that up—especially as those use cases will be diluted with the (presumably lighter) tracking use case in the above graph.

While some of the above examples fare better or worse, it must be remembered that it's the cumulative effect of these that really impacts the performance of a website. It's rare for websites to only use one of these, so add together Google Analytics, GTM loading Facebook and Twitter Tracking, on a page with a Map and an embedded Tweet, and it really starts to add up. Sometimes it's unsurprising why your phone sometimes feels too hot to handle, or your PC fan starts going into overdrive just from surfing the web!

All this shows why Google recommends reducing the impact of embeds<sup>270</sup> (mostly their own ironically!), through the use of document ordering, lazy-loading, facades, and other techniques. However, it's really quite infuriating that some of these are not the default and that advanced techniques like these must fall on the responsibility of the website owner. The third parties highlighted here really do have the resources, and technical know-how to reduce the impact of

268. <https://twitter.com/TheRealNooshu>

269. <https://nooshu.com/blog/2021/02/06/using-puppeteer-and-squoosh-to-fix-twitter-embeds/>

270. <https://web.dev/embed-best-practices/>

using their products for everyone by default, but often choose not to. This performance section started by saying that using third parties wasn't necessarily bad for performance, but these examples show there is certainly more that some of them can do in this area!

Hopefully highlighting some of these well-known examples will cause readers to investigate the impact of third-party embeds on their own sites and ask themselves if they really are all worth it. Perhaps if we make this subject more important to the third parties, they will prioritize performance.

## Timing-Allow-Origin header prevalence

Last year we looked at the prevalence of the `timing-allow-origin` header, which allows the Resource Timing API<sup>271</sup> to be used on third-party requests. Without this HTTP header, the information available to on-page performance monitoring tools for third-party requests is restricted for security and privacy reasons. However, for static requests, third parties that allow this header enable greater transparency into the loading performance of their resources.

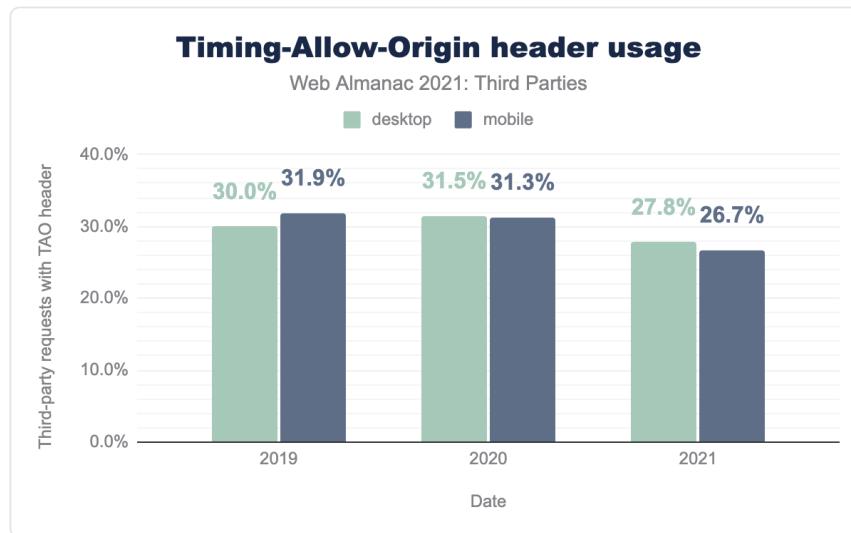


Figure 7.22. Timing-Allow-Origin header usage.

Looking at the usage over the last three Web Almanac years, usage has dropped considerably this year. Digging deeper into the data showed a 33% drop in Facebook requests. Given that they supported this header and are widely used, this explains most of this drop. Interestingly, the number of pages with Facebook usage actually increased, but it looks like Facebook have

<sup>271.</sup> [https://developer.mozilla.org/docs/Web/API/Resource\\_Timing\\_API/Using\\_the\\_Resource\\_Timing\\_API](https://developer.mozilla.org/docs/Web/API/Resource_Timing_API/Using_the_Resource_Timing_API)

changed their embed to make fewer requests in the last year and, given their prevalence, that's made quite a dent on the usage of the `timing-allow-origin` header. Ignoring that, usage of this header has basically stayed stable, which is a bit disappointing given the focus on performance with the ranking impact of the Core Web Vitals<sup>272</sup>.

## Security and Privacy

Measuring the security and privacy impact of using third parties is more difficult. Undoubtedly, giving access to third parties increases risks on both security and privacy, and then giving access to run scripts—which we've shown to be the most prevalent type—effectively gives full access to the website. However, the entire intent of third-party resources is to allow them to be seamlessly used on the sites, meaning restricting this will limit the very functionality they are being used for.

### Security

Sites themselves can reduce the risk of using third parties in a number of ways: restricting access to cookies<sup>273</sup> with the `HttpOnly` attribute, so they cannot be accessed by JavaScript, and through appropriate use of `SameSite` attributes. These are explored more in the Security chapter so we will not delve further into them here.

Another security feature that can make third-party resources safer is the use of Subresource Integrity<sup>274</sup> (SRI), which is enabled by adding a cryptographic hash of a resource to the `<link>` or `<script>` element loading the resource. This hash is then checked by the browser to ensure that the content downloaded is exactly what is expected. However, the varying nature of third-party resources could mean that this introduces more risks than it solves, with sites breaking when resources are intentionally updated by the third party. If content really is static, then it can be self-hosted, removing the need of SRI. So, while many people recommend SRI, this author remains unconvinced that it really offers the security benefits that proponents claim.

One of the best ways sites can reduce the security risk of any third-party content coming onto their site—from either third-party resource use, or even user-generated content—is with a robust Content Security Policy<sup>275</sup> (CSP). This is an HTTP header sent with the original website that tells the browser exactly what resources can and cannot be loaded and by whom. It is a more advanced technique that few sites use, according to the Security chapter, and we'll leave it to them to analyze CSP usage, but what is worth covering here is that one of the reasons for the

---

<sup>272</sup>. <https://developers.google.com/search/blog/2020/11/timing-for-page-experience>

<sup>273</sup>. [https://developer.mozilla.org/docs/Web/HTTP/Cookies#restrict\\_access\\_to\\_cookies](https://developer.mozilla.org/docs/Web/HTTP/Cookies#restrict_access_to_cookies)

<sup>274</sup>. [https://developer.mozilla.org/docs/Web/Security/Subresource\\_Integrity](https://developer.mozilla.org/docs/Web/Security/Subresource_Integrity)

<sup>275</sup>. <https://developer.mozilla.org/docs/Web/HTTP/CSP>

lack of uptake may be third parties. In this author's experience, very few third parties publish CSP information with the exact requirements that sites must add to their policy to use the third party without issue. Worse still is that others are incompatible with a secure CSP. Some third parties use inline script elements or change domains without notification, which breaks that functionality for sites using CSP until they update their policy. Google Ads is another example which, through the use of a different domain per country<sup>276</sup>, makes it difficult to really lock down CSP.

It is difficult enough to set up CSP in the first place for the parts of the site in your control, without the added complexity of third parties making it even more difficult for things outside of your control! Third parties really should get better at supporting CSP to make it easier for sites to reduce the risk of using them.

## Privacy

The privacy implications of using third parties is something we will again leave to the Privacy chapter dedicated to this topic, but what should already be apparent from the above analysis are the following two things that majorly impact the privacy of web users:

- The prevalence of third-party usage on the web at just shy of 95% of websites.
- The dominance of particular third parties, like Google and Facebook, who are not known for being on the side of privacy.

Of course, one of the major reasons for using third parties on your site is for tracking for advertisement purposes, which by its very nature is not going to be in the best privacy interests of your visitors. Alternatives to this pervasive tracking, which is basically only possible by the use of third parties, have been suggested such as Google's Privacy Sandbox and FLoC initiative<sup>277</sup> but have, so far, failed to gain sufficient traction across the wider ecosystem.

What is perhaps more concerning is the tracking that can occur without website users and owners being aware. There is the old adage that if you're not paying for a product or service, then you are the product. Many third parties give away their product for "free", which for most means they are monetizing it in some other way—usually at the expense of your visitors' privacy!

Adoption of newer technologies like `feature-policy` and `permission-policy` can restrict the usage of certain functionalities of the browser, such as microphones and video cameras. These can reduce the privacy and security risks; though many of these will usually be

276. <https://stackoverflow.com/questions/34361383/google-adwords-csp-content-security-policy-img-src>

277. <https://blog.google/products/ads-commerce/2021-01-privacy-sandbox/>

secured behind a browser prompt to ensure they are not silently activated. Google is also working on a Privacy Budget proposal<sup>278</sup> to limit the privacy impact of web browser, though others remain skeptical of their work in this space<sup>279</sup>. All in all, adding privacy controls seems to be swimming against the tide given the intent of many third-party resources.

## Conclusion

Third parties are integral to the web. In many ways they are the web; without the prevalence of third parties, websites would be harder to build and less feature rich. As mentioned at the beginning, interconnectedness is at the very heart of the web, and third parties are the natural extension of this. Our analysis has shown that third parties are more prevalent than ever—sites without them are very much the exception!

However, using third parties is not without risks and in this chapter, we have explored the performance impact of third parties and discussed the potential security and privacy risks of using them on your site.

There are consequences to needlessly loading up your website with every third-party tool, widget, tracker and whatever else you can think of. Site owners have a responsibility to look at the impact of all that third-party content and decide if the functionality is worth that potential impact.

It's easy to get sucked into the negative however, so to finish off the chapter, let's look back at the positives. There is a reason that third parties are so prevalent and they are (usually!) used out of choice. Sharing is what the web is about and so third parties are very much in the spirit of the web. It's amazing what functionality we web developers have at our disposal and how easy it is to add them to our sites. Hopefully this chapter has opened your eyes to give a little more thought to making sure you fully understand the deal you're making when you do that.

278. <https://github.com/blslassey/privacy-budget>  
279. <https://blog.mozilla.org/en/mozilla/google-privacy-budget-analysis/>

## Author

---



### Barry Pollard

[@tunetheweb](https://twitter.com/tunetheweb) [O tunetheweb](https://www.facebook.com/tunetheweb) [In tunetheweb](https://www.linkedin.com/company/tunetheweb/) [⊕ https://www.tunetheweb.com](https://www.tunetheweb.com)

Barry Pollard is a software developer and author of the Manning book HTTP/2 in Action<sup>280</sup>. 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](https://www.tunetheweb.com).

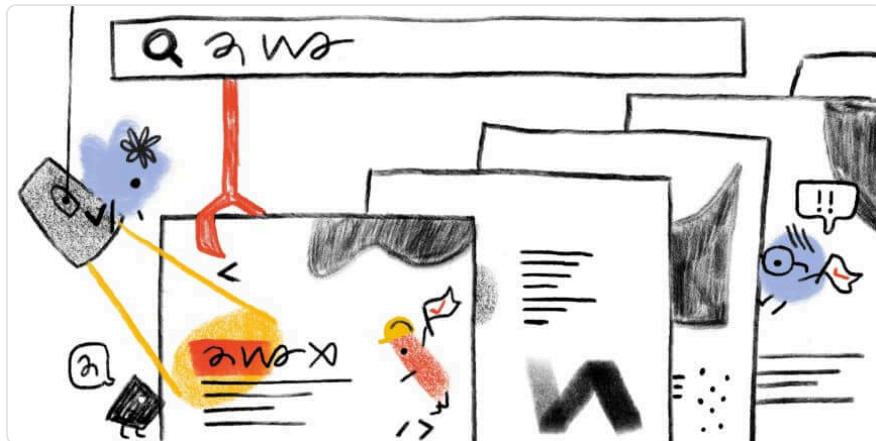
---

---

<sup>280.</sup> <https://www.manning.com/books/http2-in-action>

## Part II Chapter 8

# SEO



*Written by Patrick Stox, Tomek Rudzki, and Ian Lurie  
Reviewed by Fili Wiese, Rob Teitelman, and Jamie Indigo  
Analyzed by JR Oakes and Ruth Everett  
Edited by Barry Pollard*

## Introduction

SEO (Search Engine Optimization) is the practice of optimizing a website or web page to increase the quantity and quality of its traffic from a search engine's organic results.

SEO is more popular than ever and has seen huge growth over the last couple years as companies sought new ways to reach customers. SEO's popularity has far outpaced other digital channels.



*Figure 8.1. Google Trends comparison of SEO versus pay-per-click, social media marketing, and email marketing.*

The purpose of the SEO chapter of the Web Almanac is to analyze various elements related to optimizing a website. In this chapter, we'll check if websites are providing a great experience for users and search engines.

Many sources of data were used for our analysis including Lighthouse<sup>281</sup>, the Chrome User Experience Report (CrUX)<sup>282</sup>, as well as raw and rendered HTML elements from the HTTP Archive<sup>283</sup> on mobile and desktop. In the case of the HTTP Archive and Lighthouse, the data is limited to the data identified from websites' homepages only, not site-wide crawls. Keep that in mind when drawing conclusions from our results. You can learn more about the analysis on our Methodology page.

Read on to find out more about the current state of the web and its search engine friendliness.

## Crawlability and Indexability

To return relevant results to these user queries, search engines have to create an index of the web. The process for that involves:

1. **Crawling** - search engines use web crawlers, or spiders, to visit pages on the internet. They find new pages through sources such as sitemaps or links between pages.

281. <https://developers.google.com/web/tools/lighthouse/>

282. <https://developers.google.com/web/tools/chrome-user-experience-report>

283. <https://httparchive.org/>

2. **Processing** - in this step search engines may render the content of the pages. They will extract information they need like content and links that they will use to build and update their index, rank pages, and discover new content.
3. **Indexing** - Pages that meet certain indexability requirements around content quality and uniqueness will typically be indexed. These indexed pages are eligible to be returned for user queries.

Let's look at some issues that may impact crawlability and indexability.

### **robots .txt**

`robots.txt` is a file located in the root folder of each subdomain on a website that tells robots such as search engine crawlers where they can and can't go.

81.9% of websites make use of the `robots.txt` file (mobile). Compared with previous years (72.2% in 2019 and 80.5% in 2020), that's a slight improvement.

Having a `robots.txt` is not a requirement. If it's returning a 404 not found, Google assumes that every page on a website can be crawled. Other search engines may treat this differently.



Figure 8.2. Breakdown of `robots.txt` status codes.

Using `robots.txt` allows website owners to control search engine robots. However, the data showed that as many as 16.5% of websites have no `robots.txt` file.

Websites may have misconfigured `robots.txt` files. For example, some popular websites were (presumably mistakenly) blocking search engines. Google may keep these websites indexed for a period of time, but eventually their visibility in search results will be lessened.

Another category of errors related to `robots.txt` is accessibility and/or network errors, meaning the `robots.txt` exists but cannot be accessed. If Google requests a `robots.txt` file and gets such an error, the bot may stop requesting pages for a while. The logic behind this is that search engines are unsure if a given page can or cannot be crawled, so it waits until `robots.txt` becomes accessible.

~0.3% of websites in our dataset returned either 403 Forbidden or 5xx. Different bots may handle these errors differently, so we don't know exactly what Googlebot may have seen.

The latest information available from Google, from 2019<sup>284</sup> is that as many as 5% of websites were temporarily returning 5xx on robots.txt, while as many as 26% were unreachable.



Figure 8.3. Breakdown of robots.txt status codes Googlebot encountered.

Two things may cause the discrepancy between the HTTP Archive and Google data:

1. Google presents data from 2 years back while the HTTP Archive is based on recent information, or
2. The HTTP Archive focuses on websites that are popular enough to be included in the CrUX data, while Google tries to visit all known websites.

<sup>284</sup>. <https://www.youtube.com/watch?v=JvYh1oe5Zx0&t=315s>

**robots.txt size**

Figure 8.4. `robots.txt` size distribution.

Most robots.txt files are fairly small, weighing between 0-100 kb. However, we did find over 3,000 domains that have a robots.txt file size over 500 KiB which is beyond Google's max limit. Rules after this size limit will be ignored.



Figure 8.5. `robots.txt` user-agent usage.

You can declare a rule for all robots or specify a rule for specific robots. Bots usually try to follow the most specific rule for their user-agents. `User-agent: Googlebot` will refer to Googlebot only, while `User-agent: *` will refer to all bots that don't have a more specific rule.

We saw two popular SEO-related robots: `mj12bot` (Majestic) and `ahrefsbot` (Ahrefs) in the top 5 most specified user agents.

### `robots.txt` search engine breakdown

User-agent	Desktop	Mobile
Googlebot	3.3%	3.4%
Bingbot	2.5%	3.4%
Baiduspider	1.9%	1.9%
Yandexbot	0.5%	0.5%

Figure 8.6. `robots.txt` search engine breakdown.

When looking at rules applying to particular search engines, Googlebot was the most

referenced appearing on 3.3% of crawled websites.

Robots rules related to other search engines, such as Bing, Baidu, and Yandex, are less popular (respectively 2.5%, 1.9%, and 0.5%). We did not look at what rules were applied to these bots.

## Canonical tags

The web is a massive set of documents, some of which are duplicates. To prevent duplicate content issues, webmasters can use canonical tags to tell search engines which version they prefer to be indexed. Canonicals also help to consolidate signals such as links to the ranking page.



Figure 8.7. Canonical tag usage.

The data shows increased adoption of canonical tags over the years. For example, 2019's edition shows that 48.3% of mobile pages were using a canonical tag. In 2020's edition, the percentage grew to 53.6%, and in 2021 we see 58.5%.

More mobile pages have canonicals set than their desktop counterparts. In addition, 8.3% of mobile pages and 4.3% of desktop pages are canonicalized to another page so that they provide a clear hint to Google and other search engines that the page indicated in the canonical tag is the one that should be indexed.

A higher number of canonicalized pages on mobile seems to be related to websites using

separate mobile URLs<sup>285</sup>. In these cases, Google recommends placing a `rel="canonical"` tag pointing to the corresponding desktop URLs.

Our dataset and analysis are limited to homepages of websites; the data is likely to be different when considering all URLs on the tested websites.

## Two methods of implementing canonical tags

When implementing canonicals, there are two methods to specify canonical tags:

1. In the HTML's `<head>` section of a page
2. In the HTTP headers (via the `Link` HTTP header)

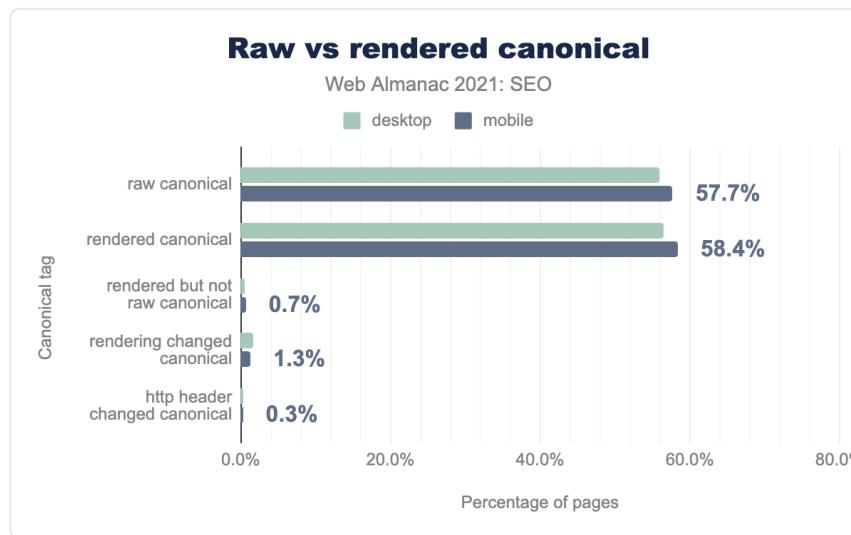


Figure 8.8. Canonical raw versus rendered usage.

Implementing canonical tags in the `<head>` of a HTML page is much more popular than using the `Link` header method. Implementing the tag in the head section is generally considered easier, which is why that usage so much higher.

We also saw a slight change (< 1%) in canonical between the raw HTML delivered, and the rendered HTML after JavaScript has been applied.

<sup>285</sup> <https://developers.google.com/search/mobile-sites/mobile-seo/separate-urls>

## Conflicting canonical tags

Sometimes pages contain more than one canonical tag. When there are conflicting signals like this, search engines will have to figure it out. One of Google's Search Advocates, Martin Splitt<sup>286</sup>, once said it causes undefined behavior<sup>287</sup> on Google's end.

The previous figure shows as many as 1.3% of mobile pages have different canonical tags in the initial HTML and the rendered version.

Last year's chapter noted that<sup>288</sup> "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."

This year's data on that conflict is even more worrisome. Pages are sending conflicting signals in 0.4% of cases on desktop and 0.3% of cases on mobile.

As the Web Almanac data only looks on homepages, there may be additional problems with pages located deeper in the architecture, which are pages more likely to be in need of canonical signals.

## Page Experience

2021 saw an increased focus on user experience. Google launched the Page Experience Update<sup>289</sup> which included existing signals, such as HTTPS and mobile-friendliness, and new speed metrics called Core Web Vitals.

<sup>286.</sup> <https://twitter.com/g33konaut>

<sup>287.</sup> <https://www.youtube.com/watch?v=bAE3L1E1Fmk&t=772s>

<sup>288.</sup> <https://almanac.httparchive.org/en/2020/seo#canonicalization>

<sup>289.</sup> <https://developers.google.com/search/blog/2020/11/timing-for-page-experience>

## HTTPS

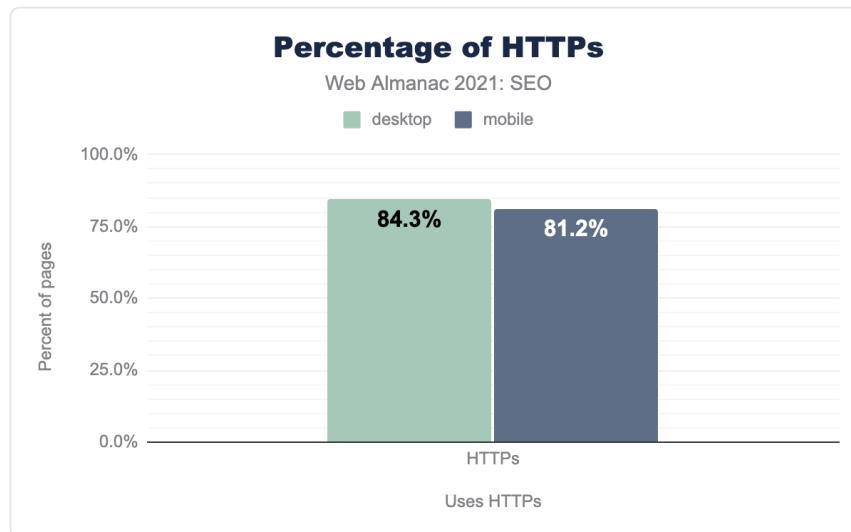


Figure 8.9. Percentage of Desktop and Mobile pages served with HTTPS.

Adoption of HTTPS is still increasing. HTTPS was the default on 81.2% of mobile pages and 84.3% of desktop pages. That's up nearly 8% on mobile websites and 7% on Desktop websites year over year.

## Mobile-friendliness

There's a slight uptick in mobile-friendliness this year. Responsive design implementations have increased while dynamic serving has remained relatively flat.

Responsive design sends the same code and adjusts how the website is displayed based on the screen size, while dynamic serving will send different code depending on the device. The `viewport` meta tag was used to identify responsive websites vs the `Vary: User-Agent` header to identify websites using dynamic serving.

**91.1%**

Figure 8.10. Percent of mobile pages using the `viewport` meta tag—a signal of mobile friendliness.

91.1% of mobile pages include the `viewport` meta tag, up from 89.2% in 2020. 86.4% of desktop pages also included the `viewport` meta tag, up from 83.8% in 2020.

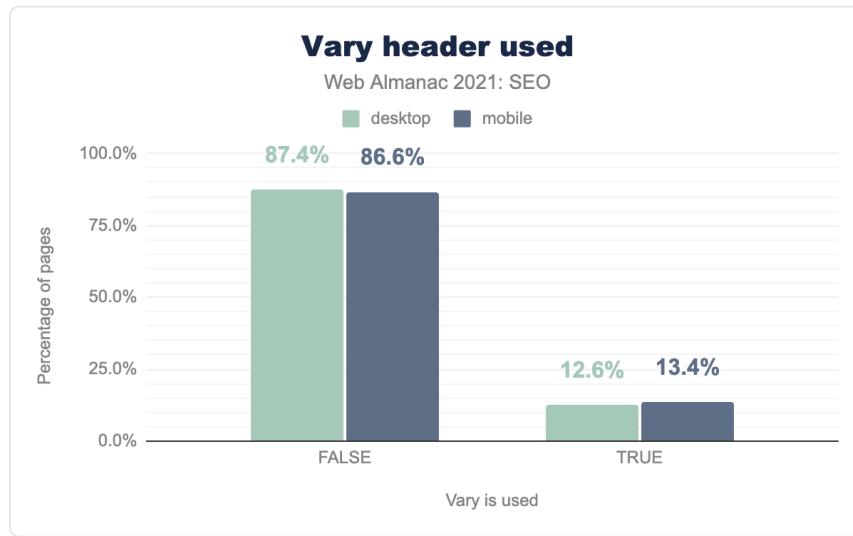


Figure 8.11. `Vary: User-Agent` header usage.

For the `Vary: User-Agent` header, the numbers were pretty much unchanged with 12.6% of desktop pages and 13.4% of mobile pages with this footprint.

# 13.5%

Figure 8.12. Percent of mobile pages not using legible font sizes.

One of the biggest reasons for failing mobile-friendliness was that 13.5% of pages did not use a legible font size. Meaning 60% or more of the text had a font size smaller than 12px<sup>290</sup> which can be hard to read on mobile.

## Core Web Vitals

Core Web Vitals are the new speed metrics that are part of Google's Page Experience signals. The metrics measure visual load with Largest Contentful Paint (LCP), visual stability with Cumulative Layout Shift (CLS), and interactivity with First Input Delay (FID).

<sup>290.</sup> <https://web.dev/font-size/>

The data comes from the Chrome User Experience Report (CrUX), which records real-world data from opted-in Chrome users.



*Figure 8.13. Core web vitals metrics trend.*

29% of mobile websites are now passing Core Web Vitals thresholds, up from 20% last year. Most websites are passing FID, but website owners seem to be struggling to improve CLS and LCP. See the Performance chapter for more on this topic.

## On-Page

Search engines look at your page's content to determine whether it's a relevant result for the search query. Other on-page elements may also impact rankings or appearance on the search engines.

## Metadata

Metadata includes `<title>` elements and `<meta name="description">` tags. Metadata can directly and/or indirectly affect SEO performance.

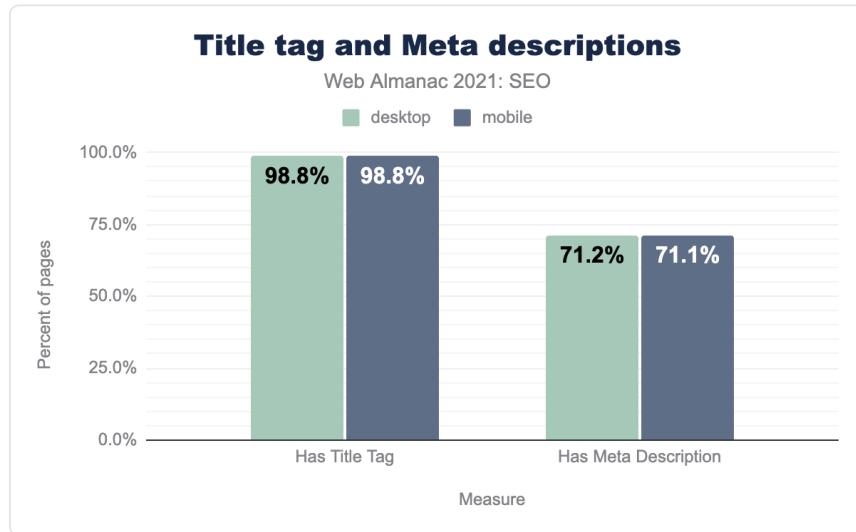


Figure 8.14. Breakdown of title and meta description usage.

In 2021, 98.8% of desktop and mobile pages had `<title>` elements. 71.1% of desktop and mobile homepages had `<meta name="description">` tags.

### `<title>` Element

The `<title>` element is an on-page ranking factor that provides a strong hint regarding page relevance and may appear on the Search Engine Results Page (SERP). In August 2021 Google started re-writing more titles in their search results<sup>291</sup>.

<sup>291</sup>. <https://developers.google.com/search/blog/2021/08/update-to-generating-page-titles>

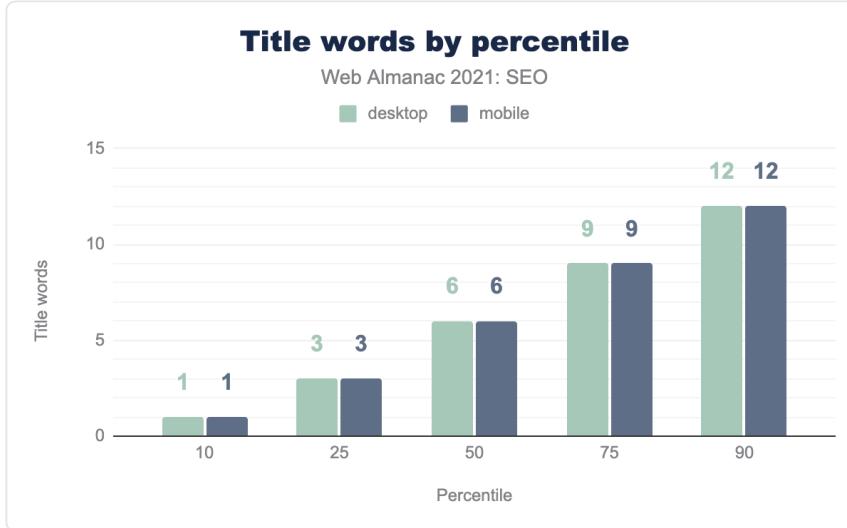


Figure 8.15. Number of words used in title elements.

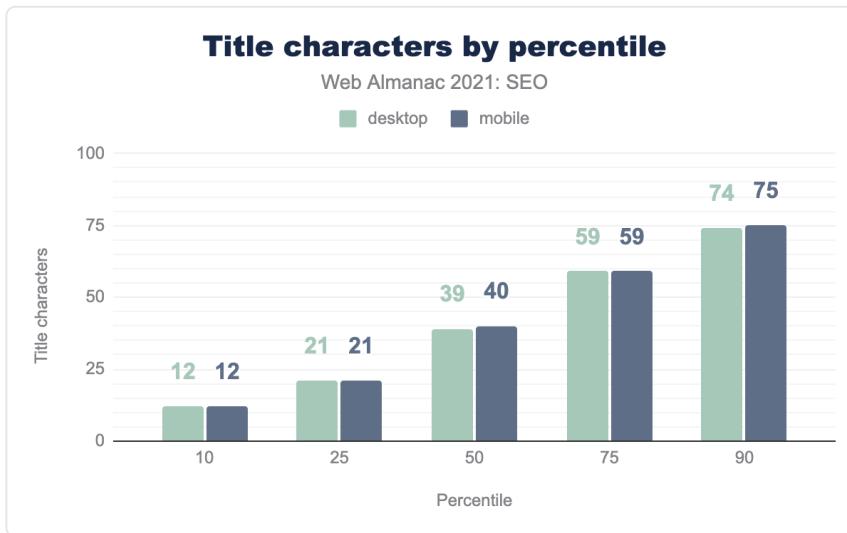


Figure 8.16. Number of characters used in title elements.

In 2021:

- The median page `<title>` contained 6 words.
- The median page `<title>` contained 39 and 40 characters on desktop and mobile,

respectively.

- 10% of pages had `<title>` elements containing 12 words.
- 10% of desktop and mobile pages had `<title>` elements containing 74 and 75 characters, respectively.

Most of these stats are relatively unchanged since last year. Reminder that these are titles on homepages which tend to be shorter than those used on deeper pages.

### Meta description tag

The `<meta name="description">` tag does not directly impact rankings. However, it may appear as the page description on the SERP.



Figure 8.17. Number of words used in meta descriptions.



Figure 8.18. Number of characters used in meta descriptions.

In 2021:

- The median desktop and mobile page `<meta name="description">` tag contained 20 and 19 words, respectively.
- The median desktop and mobile page `<meta name="description">` tag contained 138 and 127 characters, respectively.
- 10% of desktop and mobile pages had `<meta name="description">` tags containing 35 words.
- 10% of desktop and mobile pages had `<meta name="description">` tags containing 232 and 231 characters, respectively.

These numbers are relatively unchanged from last year.

## Images



Figure 8.19. Number of images on each page.

Images can directly and indirectly impact SEO as they impact image search rankings and page performance.

- 10% of pages have two or fewer `<img>` tags. That's true of both desktop and mobile.
- The median desktop page has 21 `<img>` tags while the median mobile page has 19 `<img>` tags.
- 10% of desktop pages have 83 or more `<img>` tags. 10% of mobile pages have 73 or more `<img>` tags.

These numbers have changed very little since 2020.

### Image `alt` attributes

The `alt` attribute on the `<img>` element helps explain image content and impacts accessibility<sup>292</sup>.

<sup>292</sup>. <https://almanac.httparchive.org/en/2021/accessibility>

Note that missing `alt` attributes may not indicate a problem. Pages may include extremely small or blank images which don't require an `alt` attribute for SEO (nor accessibility) reasons.

### Percentage of img alt attributes present

Web Almanac 2021: SEO

desktop mobile

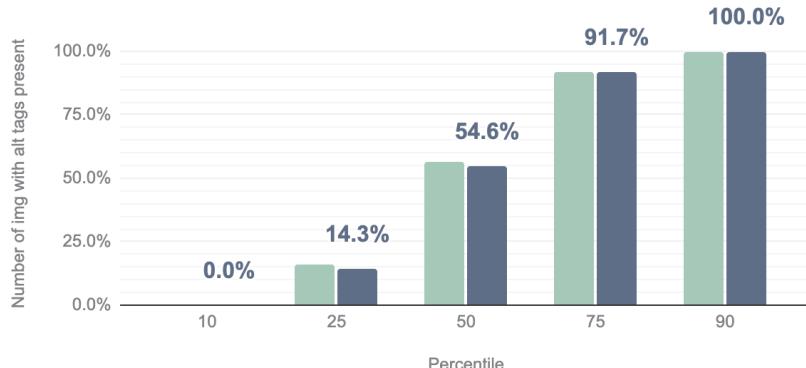


Figure 8.20. Percentage of images that contain `alt` attributes.

### Percentage of blank img alt attributes

Web Almanac 2021: SEO

desktop mobile

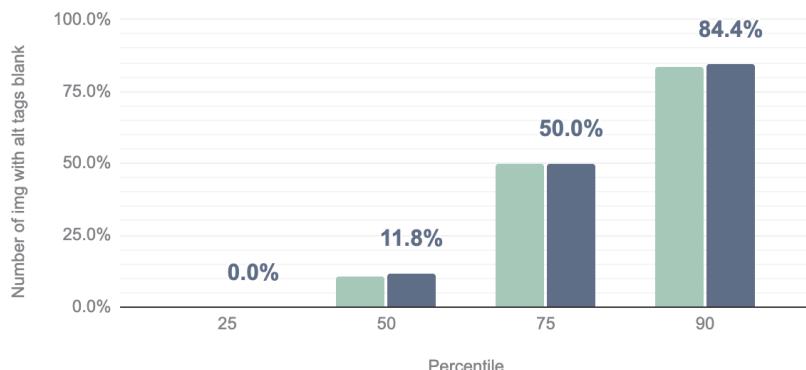


Figure 8.21. Percentage of `alt` attributes that were blank.

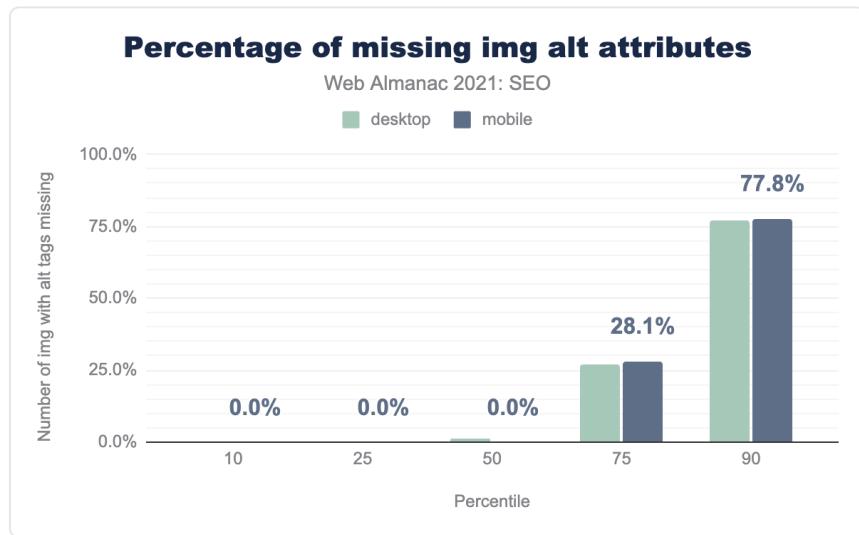


Figure 8.22. Percentage of images missing `alt` attributes.

We found that:

- On the median desktop page, 56.5% of `<img>` tags have an `alt` attribute. This is a slight increase versus 2020.
- On the median mobile page, 54.6% of `<img>` tags have an `alt` attribute. This is a slight increase versus 2020.
- However, on the median desktop and mobile pages 10.5% and 11.8% of `<img>` tags have blank `alt` attributes (respectively). This is effectively the same as 2020.
- On the median desktop and mobile pages there are zero or close to zero `<img>` tags missing `alt` attributes. This is an improvement over 2020, when 2-3% of `<img>` tags on median pages were missing `alt` attributes.

## Image `loading` attributes

The `loading` attribute on `<img>` elements affects how user agents prioritize rendering and display of images on the page. It may impact user experience and page load performance, both of which impact SEO success.

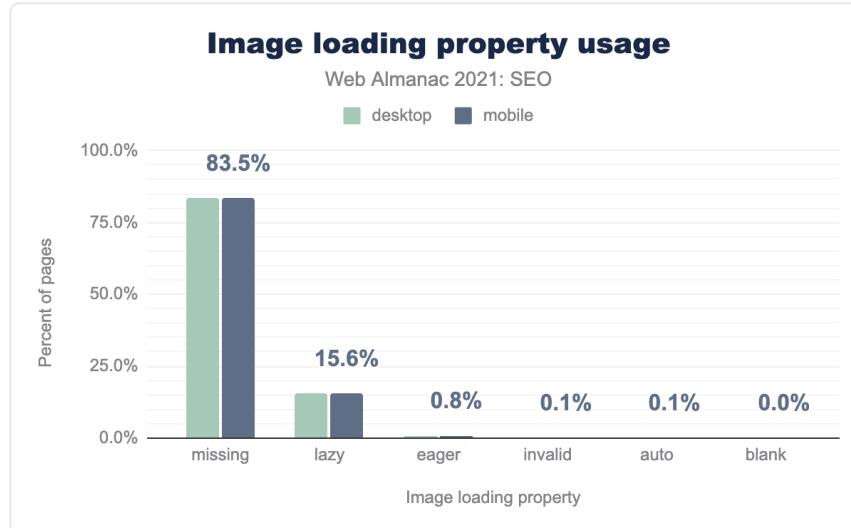


Figure 8.23. Image loading property usage.

We saw that:

- 85.5% of pages don't use any image `loading` property.
- 15.6% of pages use `loading="lazy"` which delays loading an image until it is close to being in the viewport.
- 0.8% of pages use `loading="eager"` which loads the image as soon as the browser loads the code.
- 0.1% of pages use invalid loading properties.
- 0.1% of pages use `loading="auto"` which uses the default browser loading method.

## Word count

The number of words on a page isn't a ranking factor, but the way pages deliver words can profoundly impact rankings. Words can be in the raw page code or the rendered content.

### Rendered word count

First, we look at rendered page content. *Rendered* is the content of the page after the browser

has executed all JavaScript and any other code that modifies the DOM or CSSOM.

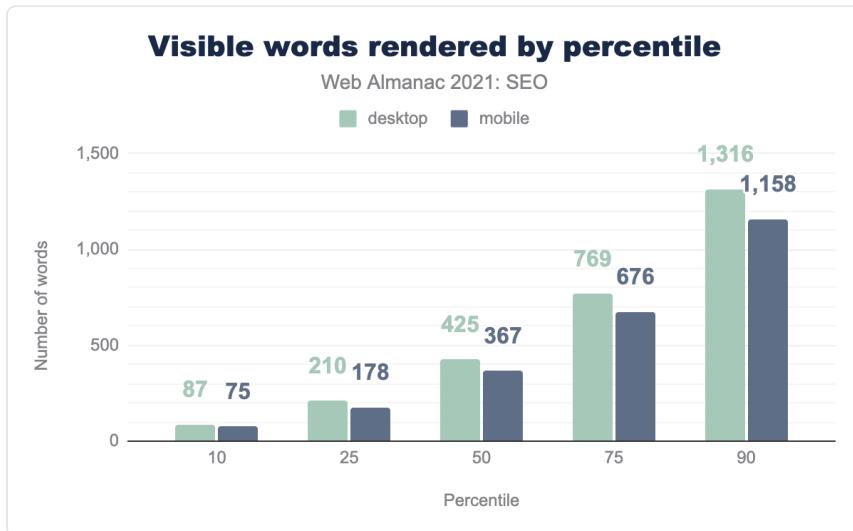


Figure 8.24. Visible words rendered by percentile.

- The median *rendered* desktop page contains 425 words, versus 402 words in 2020.
- The median *rendered* mobile page contains 367 words, versus 348 words in 2020.
- Rendered mobile pages contain 13.6% fewer words than rendered desktop pages. Note that Google is a mobile-only index. Content not on the mobile version may not get indexed.

## Raw word count

Next, we look at the raw page content. Raw is the content of the page before the browser has executed JavaScript or any other code that modified the DOM or CSSOM. It's the "raw" content delivered and visible in the source code.



Figure 8.25. Visible words raw by percentile.

- The median *raw* desktop page contains 369 words, versus 360 words in 2020.
- The median *raw* mobile page contains 321 words, versus 312 words in 2020.
- Raw mobile pages contain 13.1% fewer words than raw desktop pages. Note that Google is a mobile-only index. Content not on the mobile HTML version may not get indexed.

Overall, 15% of written content on desktop devices is generated by JavaScript and 14.3% on mobile versions.

## Structured Data

Historically, search engines have worked with unstructured data: the piles of words, paragraphs and other content that comprise the text on a page.

Schema markup and other types of structured data provide search engines another way to parse and organize content. Structured data powers many of Google's search features<sup>293</sup>.

Like words on the page, structured data can be modified with JavaScript.

<sup>293</sup>. <https://developers.google.com/search/docs/advanced/structured-data/search-gallery>



Figure 8.26. Structure data usage.

42.5% of mobile pages and 41.8% of desktop pages have structured data in the HTML.  
 JavaScript modifies the structured data on 4.7% of mobile pages and 4.5% of desktop pages.  
 On 1.7% of mobile pages and 1.4% of desktop pages structured data is added by JavaScript where it didn't exist in the initial HTML response.

## Most popular structured data formats



Figure 8.27. Breakdown of structured data formats.

There are several ways to include structured data on a page: JSON-LD, microdata, RDFa, and microformats2. JSON-LD is the most popular implementation method. Over 60% of desktop and mobile pages that have structured data implement it with JSON-LD.

Among websites implementing structured data, over 36% of desktop and mobile pages use microdata and less than 3% of pages use RDFa or microformats2.

Structured data adoption is up a bit since last year. It's used on 33.2% of pages in 2021 vs 30.6% in 2020.

## Most popular schema types



Figure 8.28. Most popular schema types.

The most popular schema types found on homepages are `WebSite`, `SearchAction`, `WebPage`. `SearchAction` is what powers the Sitelinks Search Box<sup>294</sup>, which Google can choose to show in the Search Results Page.

## `<h>` elements (headings)

Heading elements (`<h1>`, `<h2>`, etc.) are an important structural element. While they don't directly impact rankings, they do help Google to better understand the content on the page.

294. <https://developers.google.com/search/docs/advanced/structured-data/sitelinks-searchbox>

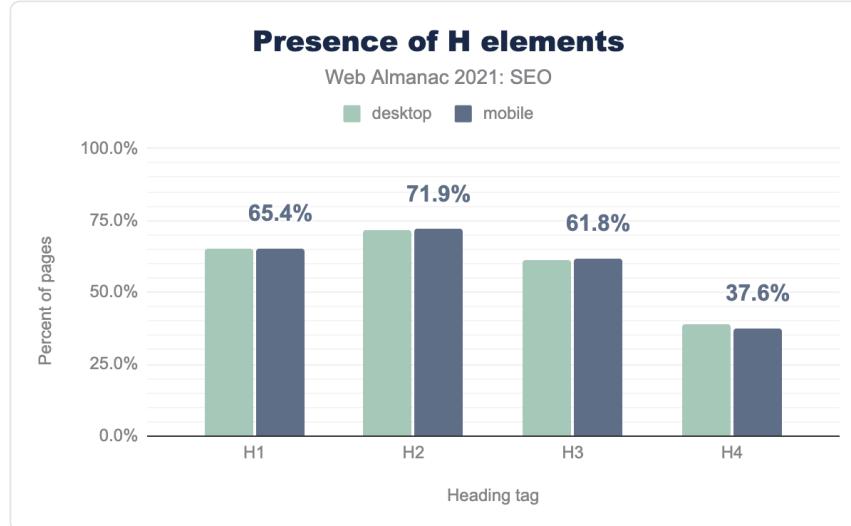


Figure 8.29. Heading element usage.

For main headings, more pages (71.9%) have `h2`s than have `h1`s (65.4%). There's no obvious explanation for the discrepancy. 61.4% of desktop and mobile pages use `h3`s and less than 39% use `h4`s.

There was very little difference between desktop and mobile heading usage, nor was there a major change versus 2020.



Figure 8.30. Non-empty heading element usage.

However, a lower percentage of pages include *non-empty* `<h>` elements, particularly `h1`. Websites often wrap logo-images in `<h1>` elements on homepages, and this may explain the discrepancy.

## Links

Search engines use links to discover new pages and to pass *PageRank* which helps determine the importance of pages.

**16.0%**

Figure 8.31. Pages using non-descriptive link texts.

On top of PageRank, the text used as a link anchor helps search engines to understand what a linked page is about. Lighthouse has a test to check if the anchor text used is useful text or if it's generic anchor text like "learn more" or "click here" which aren't very descriptive. 16% of the tested links did not have descriptive anchor text, which is a missed opportunity from an SEO perspective and also bad for accessibility.

## Internal and external links



*Figure 8.32. Internal links from homepages.*

Internal links are links to other pages on the same site. Pages had less links on the mobile versions compared to the desktop versions.

The data shows that the median number of internal links on desktop is 16% higher than mobile, 64 vs 55 respectively. It's likely this is because developers tend to minimize the navigation menus and footers on mobile to make them easier to use on smaller screens.

The most popular websites (the top 1,000 according to CrUX data) have more outgoing internal links than less popular websites. 144 on desktop vs. 110 on mobile, over two times higher than the median! This may be because of the use of mega-menus on larger sites that generally have more pages.

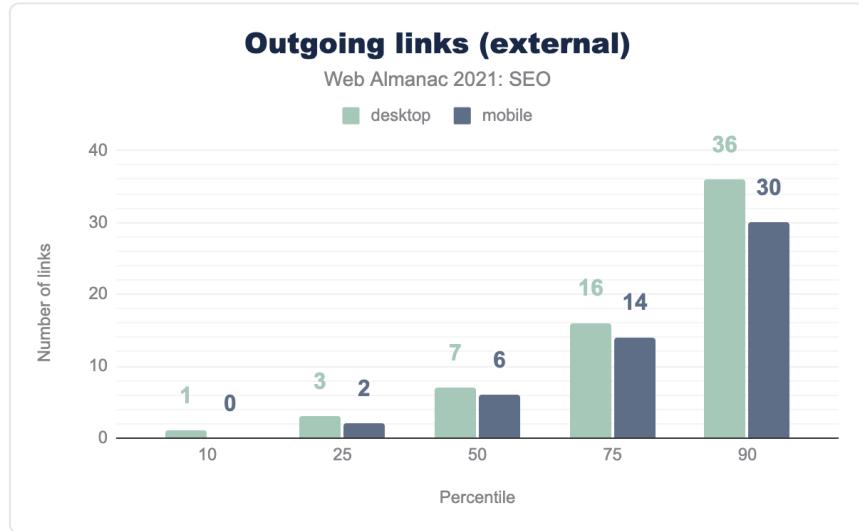


Figure 8.33. External links from homepages.

External links are links from one website to a different site. The data again shows fewer external links on the mobile versions of the pages.

The numbers are nearly identical to 2020. Despite Google rolling out mobile first indexing this year, websites have not brought their mobile versions to parity with their desktop versions.

## Text and image links



Figure 8.34. Text links from homepages.

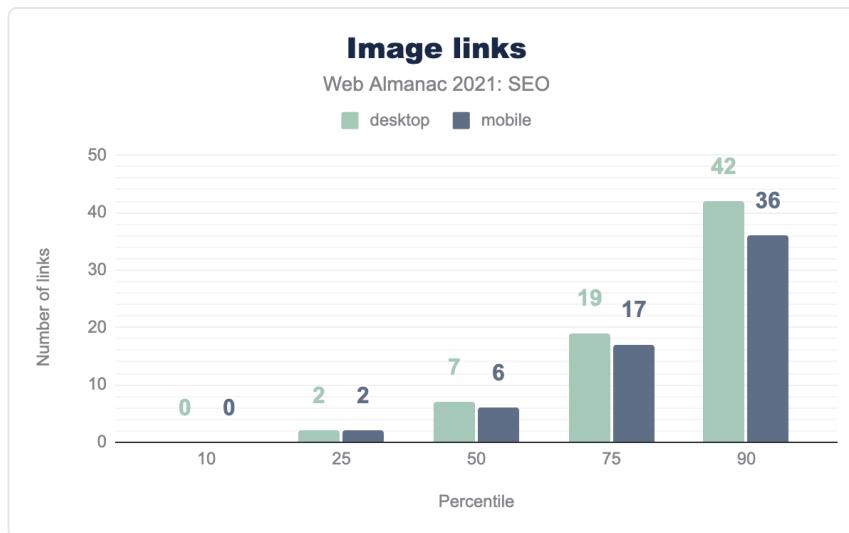


Figure 8.35. Image links from homepages.

While a significant portion of links on the web are text based, a portion also link images to other pages. 9.2% of links on desktop pages and 8.7% of links on mobile pages are image links. With image links, the `alt` attributes set for the image act as anchor text to provide additional

context on what the pages are about.

## Link attributes

In September of 2019, Google introduced attributes<sup>295</sup> 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<sup>296</sup>. The new attributes, `rel=ugc` and `rel=sponsored`, add additional information to the links.

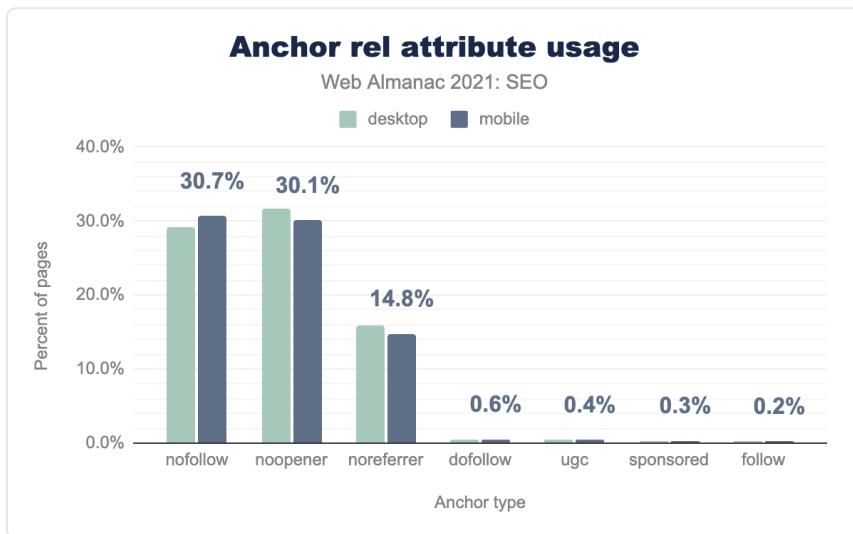


Figure 8.36. Rel attribute usage.

The new attributes are still fairly rare, at least on homepages, with `rel="ugc"` appearing on 0.4% of mobile pages and `rel="sponsored"` appearing on 0.3% of mobile pages. It's likely these attributes are seeing more adoption on pages that aren't homepages.

`rel="follow"` and `rel=dofollow` appear on more pages than `rel="ugc"` and `rel="sponsored"`. While this is not a problem, Google ignores `rel="follow"` and `rel="dofollow"` because they aren't official attributes.

`rel="nofollow"` was found on 30.7% of mobile pages, similar to last year. With the attribute used so much, it's no surprise that Google has changed `nofollow` to a hint—which means they can choose whether or not they respect it.

295. <https://webmasters.googleblog.com/2019/09/evolving-nofollow-new-ways-to-identify.html>

296. <https://googleblog.blogspot.com/2005/01/preventing-comment-spam.html>

## Accelerated Mobile Pages (AMP)

2021 saw major changes in the Accelerated Mobile Pages (AMP) ecosystem. AMP is no longer required for the Top Pages carousel, no longer required for the Google News app, and Google will no longer show the AMP logo next to AMP results in the SERP<sup>297</sup>.



Figure 8.37. AMP attribute usage.

However, AMP adoption continued to increase in 2021. 0.09% of desktop pages now include the AMP attribute vs 0.22% for mobile pages. This is up from 0.06% on desktop pages and 0.15% on mobile pages in 2020.

## Internationalization

*If you have multiple versions of a page for different languages or regions, tell Google about these different variations. Doing so will help Google Search point users to the most appropriate version of your page by language or region.*

– Google SEO documentation<sup>298</sup>

297. <https://developers.google.com/search/blog/2021/04/more-details-page-experience#details>

298. <https://developers.google.com/search/docs/advanced/crawling/localized-versions>

To let search engines know about localized versions of your pages, use `hreflang` tags.

`hreflang` attributes are also used by Yandex<sup>299</sup> and Bing (to some extent<sup>300</sup>).

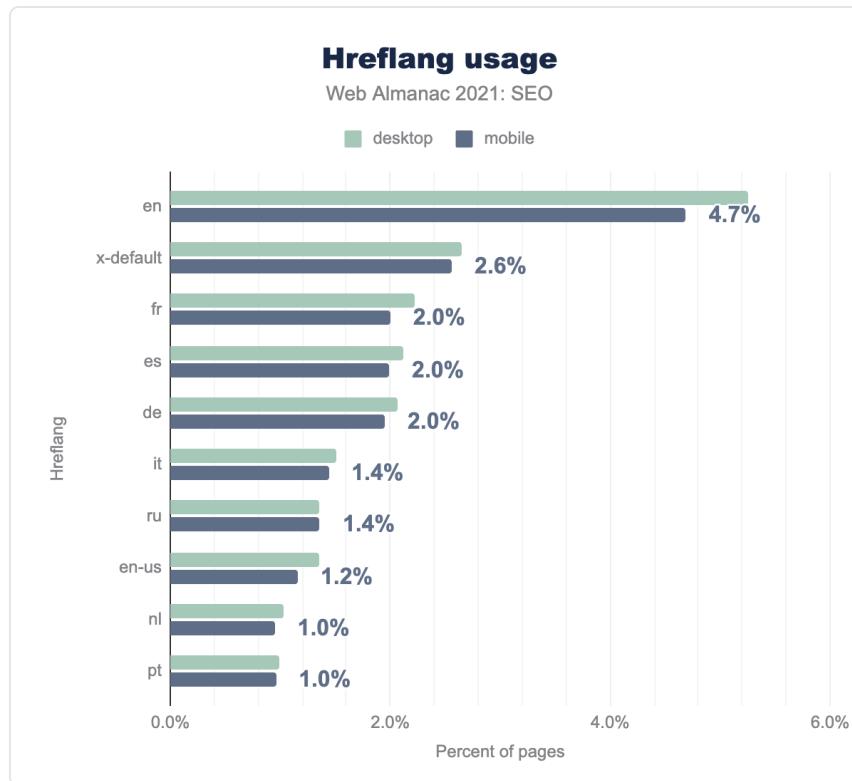


Figure 8.38. Top hreflang tag attributes chart.

9.0% of desktop pages and 8.4% of mobile pages use the hreflang attribute.

There are three ways of implementing `hreflang` information: in HTML `<head>` elements, `Link` headers, and with XML sitemaps. This data does not include data for XML sitemaps.

The most popular hreflang attribute is "en" (English version). 4.75% of mobile homepages use it and 5.32% of desktop homepages.

`x-default` (also called the fallback version) is used in 2.56% of cases on mobile. Other popular languages addressed by `hreflang` attributes are French and Spanish.

299. <https://yandex.com/support/webmaster/yandex-indexing/locale-pages.html>

300. <https://twitter.com/facan/status/1304120691172601856>

For Bing, `hreflang` is a “far weaker signal” than the `content-language` header.

As with many other SEO parameters, `content-language` has multiple implementation methods including:

1. HTTP server response
2. HTML tag

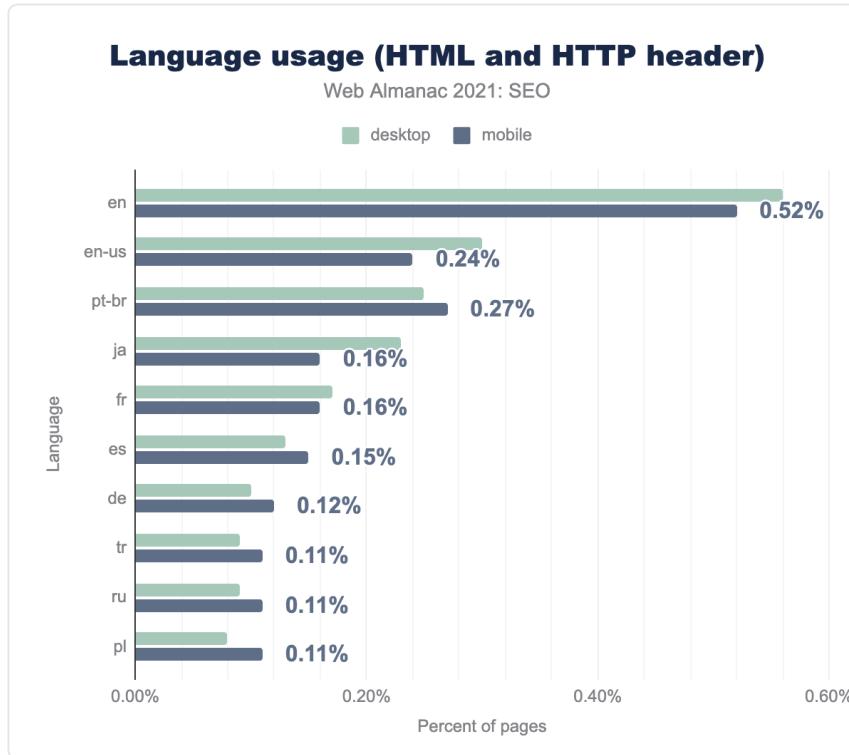


Figure 8.39. Language usage (HTML and HTTP header).

Using an HTTP server response is the most popular way of implementing `content-language`. 8.7% of websites use it on desktop while 9.3% on mobile.

Using the HTML tag is less popular, with `content-language` appearing on just 3.3% of mobile websites.

## Conclusion

Websites are slowly improving from an SEO perspective. Likely due to a combination of websites improving their SEO and the platforms hosting websites also improving. The web is a big and messy place so there's still a lot to do, but it's nice to see consistent progress.

### Authors



#### Patrick Stox

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

Patrick is Product Advisor, Technical SEO, and Brand Ambassador at Ahrefs<sup>301</sup>. He's an organizer for the Raleigh SEO Meetup<sup>302</sup> (the most successful SEO Meetup in the US), the Beer and SEO Meetup<sup>303</sup>, and the Raleigh SEO Conference<sup>304</sup>. He also runs a Technical SEO Slack group and is a moderator for /r/TechSEO on Reddit<sup>305</sup>. Patrick also likes to share random SEO knowledge in Twitter threads he calls Uncommon SEO Knowledge. He's a well-known conference speaker, industry blogger (mostly on the Ahref's blog<sup>306</sup> these days), judge of search awards, and he helped define the role of Search Marketing Strategist for the US Department of Labor.



#### Tomek Rudzki

@TomekRudzki Tomek3c <https://tomekseo.com/>

Tomek is the Head of Research and Development at Onely<sup>307</sup>. He's also building ZipTie<sup>308</sup>, a product aiming to help website owners get more content indexed by Google. In his spare time, he enjoys hiking and playing poker.

301. <https://ahrefs.com/>  
 302. <https://www.meetup.com/RaleighSEO/>  
 303. <https://www.meetup.com/beerandseo/>  
 304. <https://raleighseomeetup.org/conference/>  
 305. <https://www.reddit.com/r/TechSEO>  
 306. <https://ahrefs.com/blog/>  
 307. <http://onely.com/>  
 308. <https://www.ziptie.dev/>



## Ian Lurie

Twitter: @ianlurie | GitHub: wrtnwrd | Website: <https://www.ianlurie.com>

Ian is a marketing consultant, SEO, speaker, and recovering agency founder. He founded Portent, a digital marketing agency, in 1995, and sold it to Clearlink in 2017. He's now on his own, consulting for brands<sup>309</sup> he loves and speaking at conferences<sup>310</sup> that provide Diet Coke. He's also trying to become a professional Dungeons & Dragons player, but it hasn't panned out. You can find him pedaling his bike up Seattle's ridiculous hills.

---

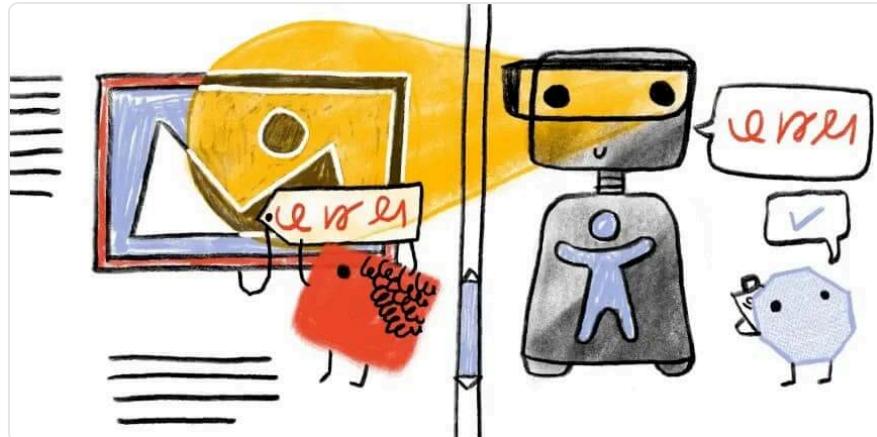
---

<sup>309.</sup> <https://www.ianlurie.com/digital-marketing-consulting/>

<sup>310.</sup> <https://www.ianlurie.com/speaking/>

# Part II Chapter 9

# Accessibility



**Written by Alex Tait, Scott Davis, Olu Niyi-Awosusi, Gary Wilhelm, and Katriel Paige**

**Reviewed by Eric Bailey, Cassey Lottman, Shaina Hantsis, Estelle Weyl, Gigi Rajani, and Carlie Dixon**

**Analyzed by David Fox**

**Edited by Barry Pollard**

## Introduction

Every year the internet grows—as of January 2021 there are 4.66 billion active internet users<sup>311</sup>. Unfortunately, accessibility is not substantially improving alongside this growth as we'll see throughout this chapter. As our reliance on internet solutions increases, so does the alienation of people who do not have equal access to the web.

2021 marked the second year of the ongoing COVID-19 pandemic. It is apparent that the disabled population is increasing as a result of long-term effects from COVID -19<sup>312</sup>. In tandem with the long-term health effects of COVID-19, society as a whole has become increasingly dependent on digital services as a result of the pandemic. Everyone is spending more time online and completing more essential activities online as well. According to the Statistics Canada Internet Use Survey<sup>313</sup>, “75% of Canadians 15 years of age and older engaged in various

311. <https://www.statista.com/statistics/617136/digital-population-worldwide/>

312. <https://www.scientificamerican.com/article/a-tsunami-of-disability-is-coming-as-a-result-of-lsquo-long-covid-rsquo/>

313. <https://www150.statcan.gc.ca/n1/pub/45-28-0001/2021001/article/00027-eng.htm>

Internet-related activities more often since the onset of the pandemic".

Products and services are also rapidly shifting online as a result of the pandemic. According to this McKinsey report<sup>314</sup>, "Perhaps more surprising is the speedup in creating digital or digitally enhanced offerings. Across regions, the results suggest a seven-year increase, on average, in the rate at which companies are developing these [online] products and services."

Web accessibility is about giving complete access to all aspects of an interface to people with disabilities by achieving feature and information parity. A digital product or website is simply not complete if it is not usable by everyone. If a digital product excludes certain disabled populations, this is discrimination and potentially grounds for fines and/or lawsuits. Last year lawsuits related to the Americans with Disabilities Act were up 20%<sup>315</sup>.

Sadly, year over year, we and other teams conducting analysis such as the WebAIM Million<sup>316</sup> are finding very little improvement in these metrics. The WebAIM study found that 97.4% of homepages had automatically detected accessibility failures, which is less than 1% lower than the 2020 audit.

The median overall site score for all Lighthouse Accessibility<sup>317</sup> audit data rose from 80% in 2020 to 82% in 2021. We hope that this 2% increase represents a shift in the right direction. However, these are automated checks, and this could also potentially mean that developers are doing a better job of subverting the rule engine.

Because 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 people with disabilities, is needed in order to achieve an accessible website or application.

We've split up our most interesting insights into six categories:

- Ease of reading
- Ease of page navigation
- Forms
- Media on the Web
- Supporting Assistive technology with ARIA
- Accessibility Overlays

---

314. <https://www.mckinsey.com/business-functions/strategy-and-corporate-finance/our-insights/how-covid-19-has-pushed-companies-over-the-technology-tipping-point-and-transformed-business-forver>

315. <https://info.usablenet.com/2020-report-on-digital-accessibility-lawsuits>

316. <https://webaim.org/projects/million/>

317. <https://web.dev/lighthouse-accessibility/>

We hope that this chapter, full of sobering metrics and demonstrable accessibility negligence on the Web, will inspire readers to prioritize this work and change their practices, shifting towards a more inclusive internet.

*We chose to use the person-first term “people with disabilities” throughout this chapter. We acknowledge that the identity-first term “disabled people” is preferred for many. Our choice in terminology is in no way prescriptive of which term is appropriate.*

## Ease of reading

Making content as simple and clear to read as possible is an important aspect of web accessibility. When people are unable to read the content of a page, not only are they unable to access its information, they are also prevented from being able to complete tasks such as registering for an account or making a purchase.

There are many aspects of a web page that make it easier or harder to read, including color contrast, zooming and scaling of pages, and language identification.

### Color contrast

Color contrast<sup>318</sup> refers to how easily text and other page artifacts stand out against the surrounding background. The higher the contrast, the easier it is for people to distinguish the content. The Web Content Accessibility Guidelines<sup>319</sup> (WCAG) has minimum contrast requirements for text and non-text 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 situational difficulties viewing the content, such as glare on screens in bright light.

318. <https://www.a11yproject.com/posts/2015-01-05-what-is-color-contrast/>

319. <https://www.w3.org/WAI/standards-guidelines/wcag/>



Figure 9.1. Mobile sites with sufficient color contrast.

This year we found that only 22% of sites have passing color contrast scores in Lighthouse. It is worth noting that these scans are only able to catch text-based contrast issues, as non-text content is so variable. This score has stayed about the same year over year; it was 21% in 2020 and 22% in 2019. This metric is somewhat disheartening, as catching text-based contrast issues is possible with a variety of common automated tools.

## Zooming and scaling

Users with low vision may rely on zooming and scaling the page using system settings or screen magnifying software in order to view its content, especially text. The Web Content Accessibility Guidelines require that text in particular can be resized up to at least 200%<sup>320</sup>.

Adrian Roselli<sup>321</sup> wrote a comprehensive article about the various harms caused when zooming is not enabled for users<sup>322</sup>. Many browsers now prevent developers from overriding zoom controls, but it must be avoided at the code-level, as we cannot count on every browser overriding this behavior when we consider the wide range of browser and OS usage on a global scale.

320. <https://www.w3.org/TR/UNDERSTANDING-WCAG20/visual-audio-contrast-scale.html>

321. <https://twitter.com/aardrian>

322. <https://adrianroselli.com/2015/10/dont-disable-zoom.html>



Figure 9.2. Pages with zooming and scaling disabled.

We found that 24% of desktop homepages and 29% of mobile homepages attempt to disable scaling by setting either `maximum-scale` to a value less than or equal to 1, or `user-scalable` set to `0` or `none`.

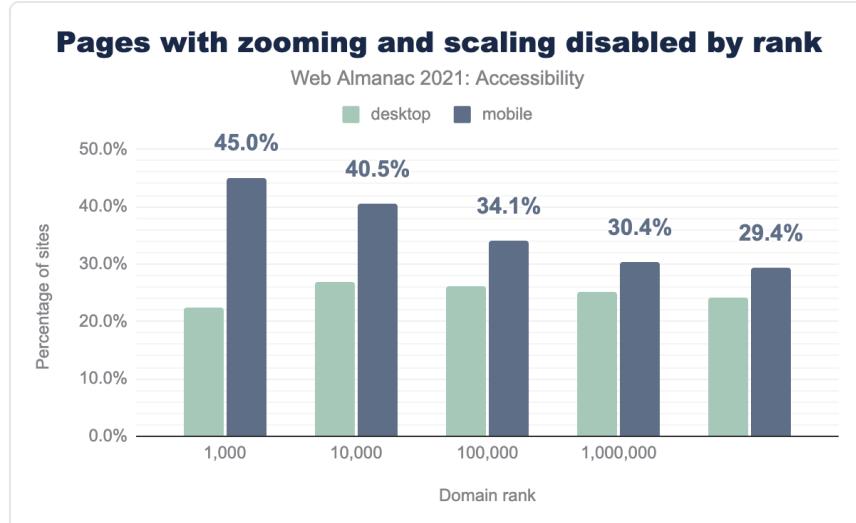


Figure 9.3. Pages with zooming and scaling disabled by rank.

When we consider the most popular sites in particular, the numbers for mobile are especially concerning. Of the top 1,000 most trafficked sites, 22% of desktop sites and 45% of mobile sites have code that attempts to disable user scaling. This may be a trend that comes from the proliferation of web applications. People need to be able to customize their web browsing experience (such as zooming and scaling) regardless of whether the content is a website or web application.

## Language identification



Figure 9.4. Mobile sites have a valid `lang` attribute.

Setting an HTML `lang` attribute allows easy translation of a page and better screen reader support, allowing some screen readers to apply the appropriate accent and inflection to the text being read. The percentage of sites with a `lang` attribute increased this year to 81% (up from 78% in 2020), and of the sites that have the attribute present, 99.7% had a valid `lang` attribute.

## Font size and line height

There is no specific requirement from the WCAG with respect to minimum font size or line height, however there is a general consensus that a base font size of 16px<sup>323</sup> or higher will help everyone with readability, especially those who have low vision. There is, however, a requirement that text can be zoomed in and resized up to 200%. Users can also set their own minimum font size at the browser level and these customized settings need to be supported.

323. <https://accessibility.digital.gov/visual-design/typography/>

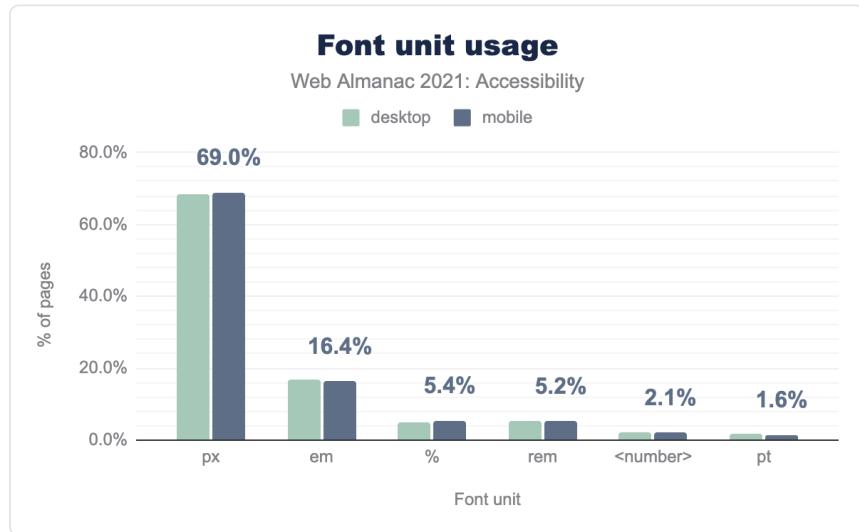


Figure 9.5. Font unit usage.

When fonts are declared in `px` units, they are static sizes. The best way to ensure that fonts scale appropriately when the browser is zoomed is to use relative units such as `em` and `rem`. We found that 68% of desktop font size declarations are set in `px`, 17% are set in `em` and 5% are set with `rem` units.

## Focus Styles

Visible focus styles are helpful for everyone but are necessary for sighted keyboard users who rely on their presence to navigate. The WCAG requires a visible focus indicator<sup>324</sup> for all interactive content.

<sup>324</sup>. <https://www.w3.org/TR/UNDERSTANDING-WCAG20/navigation-mechanisms-focus-visible.html>

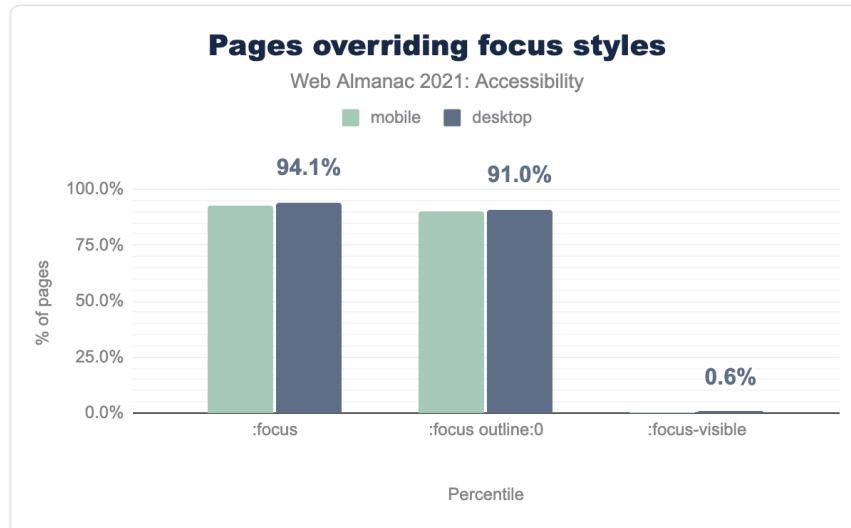


Figure 9.6. Pages overriding focus styles.

Often times, default focus indication is removed from interactive content such as buttons, form controls, and links using the CSS property `:focus { outline: none; }` or `:focus { outline: 0; }`, sometimes in conjunction with `:focus-within` and/or `:focus-visible`. We found that 91% of desktop pages have `:focus { outline: 0; }` declared. In some cases, it is removed so that a more effective custom style can be applied. Unfortunately, in many cases it is simply removed and never replaced, which can render a page unusable for keyboard users.

For more information about how to achieve accessible focus indication including some limitations of browser default focus styles, we recommend Sara Soueidan<sup>325</sup>'s article, "A guide to designing accessible, WCAG-compliant focus indicators"<sup>326</sup>.

## User preference media queries and high contrast support

The CSS Media Queries Level 5 specification<sup>327</sup>, published in 2020, introduced a collection of *User Preference Media Features* that allow a website to detect Accessibility features that a user may have configured outside of the website itself. These features are typically configured through operating system or platform preferences.

325. <https://twitter.com/SaraSoueidan>

326. <https://www.sarasoueidan.com/blog/focus-indicators/>

327. <https://www.w3.org/TR/mediaqueries-5>

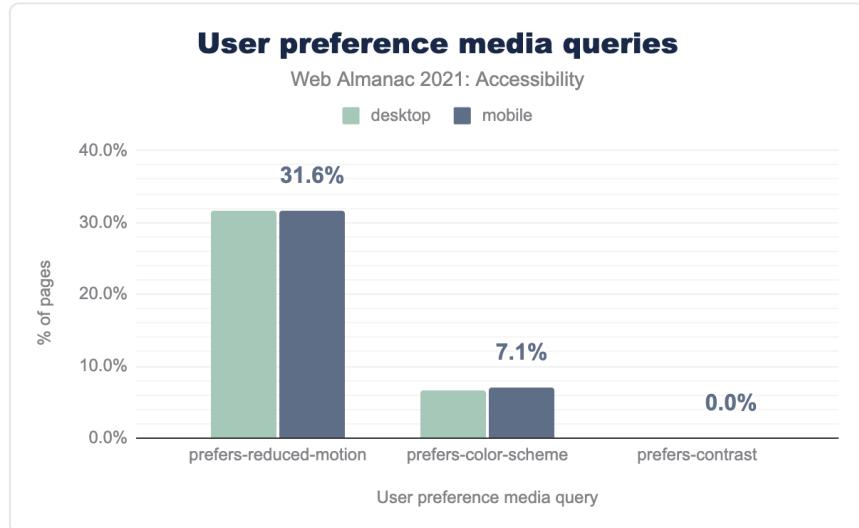


Figure 9.7. User preference media queries.

`prefers-reduced-motion` is used by web authors to replace animations or other sources of motion on the web page with a more static experience, typically by removing or replacing the content. This can help a range of people that may be distracted or otherwise triggered by rapid movement on the screen. We found that 32% of websites use the `prefers-reduced-motion` media query.

`prefers-reduced-transparency` indicates that the end user has asked the operating system to minimize or eliminate translucency and transparency effects. This affordance might be turned on by end users to help with reading comprehension or to avoid common “halo effects” that can negatively affect users with visual impairments. We do not have data on the usage of this relatively new media query.

`prefers-contrast` (`high` or `low`) suggests that the end user would prefer a high-contrast or low-contrast contrast theme. This can help with reading comprehension and eye strain. We do not have data on the usage of this relatively new media query though we found that 25% of websites use `ms-high-contrast` which is a Windows-specific approach to handling contrast preferences.

`prefers-color-scheme` (`light` or `dark`) allows a user to request light color on a dark background experience, or vice-versa. This was the earliest of the User Preference Media Queries to be introduced. This capability, commonly known as “dark mode” support, rose to prominence in 2019 after Apple standardized it<sup>328</sup> in iOS 13 and iPadOS, though it had been a

328. [https://en.wikipedia.org/wiki/Light-on-dark\\_color\\_scheme#History](https://en.wikipedia.org/wiki/Light-on-dark_color_scheme#History)

common accessibility feature for many years prior to that.

While dark mode is recognized by many developers and designers as an accessibility affordance, it is important to note that dark mode may, in fact, reduce accessibility for certain users. Some people with dyslexia or astigmatism might find light text on a dark background harder to read<sup>329</sup>, and might find that it exacerbates the halo effect. The important takeaway here is to let your user choose what works best for them. We found that 7% of websites use the `prefers-color-scheme` media query.

## Ease of page navigation

Navigating through web content is one of the fundamental ways we engage online and there are many ways this is accomplished. For some people, this could mean visually scanning a page while scrolling with a mouse. For others it might start by navigating through the headings on a page with their screen reader. Websites need to be easy to navigate so users are not left feeling lost or unable to find the content they are seeking.

### Landmarks and page structure

Landmarks are designated HTML elements or ARIA roles we can apply to other HTML elements that enable assistive technology users to quickly understand overall page structure and navigation. For example a rotor menu<sup>330</sup> can be used to navigate to different landmarks of the page, and or a skip link can be used to target the `<main>` landmark.

Before the introduction of HTML5, ARIA landmark roles were needed to accomplish this. However, we now have native HTML elements available to accomplish the majority of landmark page structure. Leveraging the native HTML landmark elements is preferable to applying ARIA roles, per the first rule of ARIA<sup>331</sup>. For more information, see the ARIA roles section of this chapter.

---

329. <https://www.boia.org/blog/dark-mode-can-improve-text-readability-but-not-for-everyone>

330. <https://webaim.org/articles/voiceover/mobile#rotor>

331. <https://www.w3.org/TR/using-aria/#rule1>

<b>HTML5 element</b>	<b>ARIA role equivalent</b>	<b>Pages with element</b>	<b>Pages with role</b>	<b>Pages with element or role</b>
<main>	role="main"	27.68%	16.90%	35.00%
<header>	role="banner"	62.13%	14.34%	63.49%
<nav>	role="navigation"	61.69%	22.79%	65.53%
<footer>	role="contentinfo"	63.35%	12.21%	64.52%

Figure 9.8. Landmark element and role usage (desktop).

The most commonly expected landmarks that the majority of web pages should have, are <main>, <header>, <nav> and <footer>. We found that only 28% of desktop pages have a native HTML <main> element, 17% of desktop pages have an element with a role="main", and 35% of pages have either.

When a page has multiple instances of the same landmark, for example, a primary site navigation and a breadcrumb secondary navigation, it is important that they each have a unique accessible name. This will help an assistive technology user to better understand which navigation landmark they have encountered. Techniques for accomplishing this are covered in Scott O'Hara<sup>332</sup>'s comprehensive article about the various landmarks and how different screen readers navigate them<sup>333</sup>.

## 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.

332. <https://twitter.com/scottohara>

333. <https://www.scottohara.me/blog/2018/03/03/landmarks.html>



*Figure 9.9. Title element statistics*

Our data shows 98% of web pages have a title. However, only 68% of those pages have a title containing four or more words, meaning that it is likely that a significant percentage of web pages do not have a unique, meaningful title that provides enough information about the content of the page.

## Secondary Navigation

Many users benefit from a secondary navigation method to help them find the content they are looking for on a website. The WCAG has a requirement that complex websites have a secondary navigation method<sup>334</sup>. One of the most common and helpful secondary navigation methods is a search mechanism. We found that 24% of all sites used a search input.

Another approach to providing a secondary navigation method is to implement a site map, which is a collection of all of the links available on a website clearly organized collection. Although we do not have any data about the presence of site maps, this technique guide from the W3C<sup>335</sup> explains what they are in detail and how to implement one effectively.

## Tabindex

`tabindex` is an attribute that can be added to elements to control whether it can be focused.

334. <https://www.w3.org/TR/UNDERSTANDING-WCAG20/navigation-mechanisms-mult-loc.html>

335. <https://www.w3.org/TR/WCAG20-TECHS/G63.html>

Depending on its value, the element can also be in the keyboard focus, or “tab” order.

A `tabindex` value of `0` allows for an element to be programmatically focusable and in keyboard focus order. Interactive content such as buttons, links, and form controls have the equivalent of a `tabindex` value of `0`, meaning they are in the keyboard focus order natively.

Custom elements and widgets that are intended to be interactive and in the keyboard focus order need an explicitly assigned `tabindex="0"`, or they will not be usable by keyboard.

If an element should be focusable but not in the keyboard focus order a `tabindex` value of `-1` (or any negative integer) can be used as a hook to enable programmatically setting focus on the element with JavaScript without adding it to the keyboard focus order. This can be helpful for cases where you’d like to assign focus, such as focusing a heading when navigating to new page within a single page application as covered by Marcy Sutton<sup>336</sup> in her post on accessible client-side routing techniques<sup>337</sup>. Placing non-interactive elements in keyboard focus order creates a confusing experience for blind and low vision users and should be avoided.

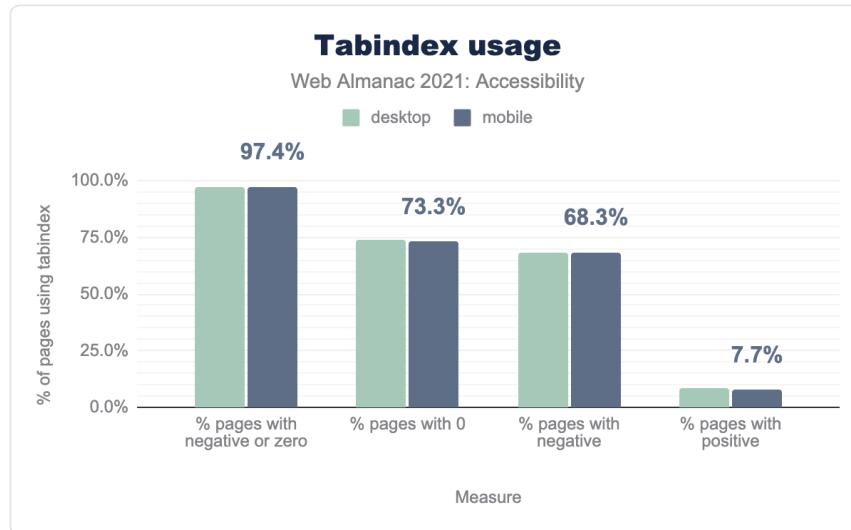
The focus order of the page should always be determined by the document flow meaning the order of the HTML elements in the document. Setting the `tabindex` to a positive integer value overrides the natural order of the page, often leading to failures of WCAG 2.4.3 - Focus Order<sup>338</sup>. Respecting the natural focus order of a page generally leads to a more accessible experience than over-engineering the keyboard focus order.

We found that 58% of desktop sites and 56% of mobile sites have some usage of the `tabindex` attribute.

336. <https://twitter.com/marcysutton>

337. <https://www.gatsbyjs.com/blog/2019-07-11-user-testing-accessible-client-routing/>

338. <https://www.w3.org/TR/UNDERSTANDING-WCAG20/navigation-mechanisms-focus-order.html>

Figure 9.10. `tabindex` usage

When we look at desktop pages that have at least one instance of the `tabindex` attribute:

- 96.9% use a value of `0`, meaning elements are focusable and being added to the keyboard focus order
- 68.2% use a negative integer, meaning elements are explicitly removed from the keyboard focus order
- 8.7% have a positive integer value, meaning the web author is trying to control the focus order rather than allowing the DOM structure to do so

While there are valid declarations for the `tabindex` attribute, incorrectly reaching for these techniques leads to common accessibility barriers for many keyboard and assistive technology users. For more information about the pitfalls of using a positive integer for `tabindex` we recommend Karl Groves<sup>339</sup> article, “Why using `tabindex` values greater than “0” is bad”.

## Skip links

Skip links help people who rely on keyboards to navigate. They enable a user to skip through sections of content that repeat across multiple pages or navigation sections and go to another destination, typically the `<main>` element of the page. Skip links are typically the first element

<sup>339</sup>. <https://twitter.com/karlgroves>

on a page and can be persistent in the UI or visibly hidden until they have keyboard focus. For example, a lot of interactive content (such as a robust navigation system full of links), can be incredibly cumbersome to tab through before reaching the main content of the screen, especially as these tend to be repeated across multiple pages.

Some websites that are very information dense have several skip links to allow users to jump to the commonly trafficked areas of the site. For example, the government of Canada's website<sup>340</sup> has "skip to main content", "skip to about government" and "switch to basic HTML version".

Skip links are considered a bypass for a block<sup>341</sup>. There is no way for us to query for all possible skip link implementations, however we found that close to 20% of desktop and mobile sites likely have a skip link. We determined this by looking for the presence of an `href="#main"` attribute on one of the first three links on the page, which is a common implementation for a skip link.

## Heading hierarchy

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.

58%

*Figure 9.11. Mobile sites passing the Lighthouse audit for properly ordered headings.*

Our audits revealed that 58% of the sites checked pass the test for properly ordered headings<sup>342</sup> that do not skip levels. Over 85% of screen reader users surveyed in 2021 by WebAIM<sup>343</sup> reported they find headings useful in navigating the web. Having headings in the correct order—ascending without skipping levels—means that assistive technology users will have the best experience.

## 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 table markup that provides a machine-readable structure so the user can effectively navigate, understand and interact with them.

340. <https://www.canada.ca/>

341. <https://www.w3.org/WAI/WCAG21/Understanding/bypass-blocks.html>

342. <https://web.dev/heading-order/>

343. <https://webaim.org/projects/screenreadersurvey9/#heading>

Tables should have a well-formatted structure with the appropriate elements and defined relationships, including a caption, appropriate headers and footers, and a corresponding header cell for every data cell. Screen reader users rely on such well-defined relationships through what is announced, so an incomplete or an incorrectly declared structure can lead to misleading or missing information.

	<b>Table sites</b>		<b>All sites</b>	
	<b>Desktop</b>	<b>Mobile</b>	<b>Desktop</b>	<b>Mobile</b>
Captioned tables	5.4%	4.6%	1.2%	1.0%
Presentational table	1.2%	0.9%	0.5%	0.4%

Figure 9.12. Accessible table usage.

## Table captions

Table captions act as a heading for the full table to provide a summary of its information. When labelling a table, the `<caption>` element is the correct semantic choice to provide the most context to a screen reader user, though it should be noted that there are also other alternative captioning techniques for tables<sup>344</sup>.

Heading elements for the full table are frequently unnecessary when a `<caption>` element has been properly implemented, and the `<caption>` element can be styled and visually positioned in a way that resembles a heading. Only 5% of desktop and mobile sites with table elements present used a `<caption>`, which is a slight increase from 2020.

## Tables for layout

The introduction of CSS methodologies such as Flexbox<sup>345</sup> and Grid<sup>346</sup> provided the capability for web developers to easily create fluid responsive layouts. Prior to this development, developers frequently used tables for layout instead of presenting data. Unfortunately, due to a combination of legacy websites and legacy development techniques, websites still exist where tables are used for layout. It is difficult to determine how widely this legacy development technique is still used.

If there is an absolute need to reach for this technique, the role of `presentation` should be

344. <https://www.w3.org/WAI/tutorials/tables/caption-summary/>

345. [https://www.w3schools.com/css/css3\\_flexbox.asp](https://www.w3schools.com/css/css3_flexbox.asp)

346. [https://www.w3schools.com/css/css\\_grid.asp](https://www.w3schools.com/css/css_grid.asp)

applied to the table such that assistive technology will ignore the table semantics. We found that 1% of desktop and mobile pages contain 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.

## Tabs

Tabs are a very common interface widget but making them accessible presents a challenge for many developers. A common pattern for accessible implementation comes from the WAI-ARIA Authoring Practices Design Patterns<sup>347</sup>. Note that the ARIA Authoring Practices document is not a specification and is meant to demonstrate idealized use of ARIA for common widgets. They should not be used in production without testing with your users.

The Authoring Practice guidelines suggest always using the `tabpanel` role in conjunction with `role="tab"`. We found that 8% of desktop pages have at least one element with a `role="tablist"`, 7% of pages have elements with a `role="tab"` and 6% of pages have elements with a `role="tabpanel"`. For more information see the ARIA roles section below.

## Captchas

Public websites regularly have two different types of visitors—humans and computers that crawl the web. To attract human visitors, websites hope to be featured prominently by search engines. Search engines, in turn, send out automated programs called web crawlers to visit websites, look around, and report their findings back to the search engine to classify and organize their content.

For example, The Web Almanac is created each year by sending out a similar kind of web crawler to gather information about roughly 8 million different websites. Authors then summarize the results for your reading pleasure.

For cases where websites want to verify that the visitor is a human, one technique web authors sometimes use is putting up a test that a human can theoretically pass, and a computer cannot. These types of “human-only” tests are called a CAPTCHA—“Completely Automated Public Turing Test, to Tell Computers and Humans Apart”.

<sup>347</sup>. <https://www.w3.org/TR/wai-aria-practices-1.1/#tabpanel>

# 10.2%

*Figure 9.13. Desktop sites using a CAPTCHA*

We found CAPTCHAs on roughly 10% of the websites visited, across both desktop and mobile sites.

CAPTCHAs present a host of potential accessibility barriers. For example, one of the most common forms of a CAPTCHA presents an image of wavy, distorted text and asks the user to decipher the text and type it in. This type of test can be difficult to solve for everyone but would likely be more difficult for people with low vision and other vision or reading related disabilities. One usability survey found that roughly 1 out of 3 users failed to successfully decipher a CAPTCHA on the first try<sup>348</sup>.

If CAPTCHAs include alt text, the test would be trivial to pass by a computer since the answer is provided as plain text. However, by not including alt text, CAPTCHAs are excluding screen readers and the blind or low vision users who use them.

For more information on the accessibility barriers that CAPTCHAs present, we recommend the W3C paper: “Inaccessibility of CAPTCHA: Alternatives to Visual Turing Tests on the Web”<sup>349</sup>.

From the paper: “It is important to acknowledge that using a CAPTCHA as a security solution is becoming increasingly ineffective... Alternative security methods, such as two-step or multi-device verification, along with emerging protocols for identifying human users with high reliability should also be carefully considered in preference to traditional image-based CAPTCHA methods for both security and accessibility reasons.”

## Forms

Forms can make or break access to the web, which increasingly means access to participation in society and essential services. Many people do their banking, grocery shopping, flight booking, appointment scheduling, and work online, as well as many other activities.

Due to the effects of the COVID-19 pandemic, millions of children went to school online in 2021. All of these services require forms to register and sign in at a minimum, and many have much more complex forms that require other sensitive information such as financial information. Inaccessible forms are discriminatory and can cause serious harm.

348. <https://baymard.com/blog/captchas-in-checkout>

349. <https://www.w3.org/TR/turingtest/>

The 2019 Click-Away Pound survey in the UK was designed “to explore the online shopping experience of people with disabilities and examine the cost to business of ignoring disabled shoppers.” It found that UK businesses missed out on over £17 billion of sales in abandoned shopping carts due to website accessibility barriers. Profit should never be the primary reason to respect the rights of people with disabilities, but the business case is very substantial.

## The `<label>` element

One of the most important ways of making HTML forms accessible is using the `<label>` element to programmatically link the short descriptive text that describes the form control<sup>350</sup>. This is typically done by matching the `for` attribute on the `<label>` element with the `id` attribute on the form control element. For example:

```
<label for="first-name">First Name</label>
<input type="text" id="first-name">
```

When a web developer fails to associate a `<label>` element with an input, they are missing out on a number of key features that they would otherwise get for free. For example, when a `<label>` is properly associated with an `<input>` field, tapping or clicking on the `<label>` automatically puts focus in the `<input>` field. This is not only a major usability win—it is also expected behavior on the web.

<sup>350.</sup> [https://developer.mozilla.org/docs/Learn/Forms/Basic\\_native\\_form\\_controls](https://developer.mozilla.org/docs/Learn/Forms/Basic_native_form_controls)

## Where inputs get their accessible names from

Web Almanac 2021: Accessibility



Figure 9.14. Where inputs get their accessible names from.

The `<label>` element was introduced with HTML 4 in 1999. Despite being available in all modern browsers for the past 20+ years, only 27% of all `<input>` elements get their accessible name from a programmatically associated label and 32% of input elements have no accessible name at all.

Most importantly, without proper accessible names, screen readers and voice to text users may not be able to target or identify the purpose of a form field. `<label>` elements associated with an input are the most robust and expected way to do this.

This is not only important when the end user is filling in the form for the first time—it is equally important if form validation finds an error with a specific field that the user must correct before they can submit the form. For example, if a user forgot to provide the expiration date for their credit card, they cannot complete their purchase. And they cannot complete their purchase if they cannot find the errant field with the missing value and understand both the purpose of the input and the steps needed to fix the error.

## The improper use of the `placeholder` attribute for labeling inputs

The `placeholder` attribute was introduced in HTML5 in 2014. Its intended use is to provide an example of the data that is expected to be provided by the user. For example, `<input type="text" id="credit-card" placeholder="1234-5678-9999-0000">` will display the placeholder as faint text in the input field that will disappear the moment the user begins typing in the field.

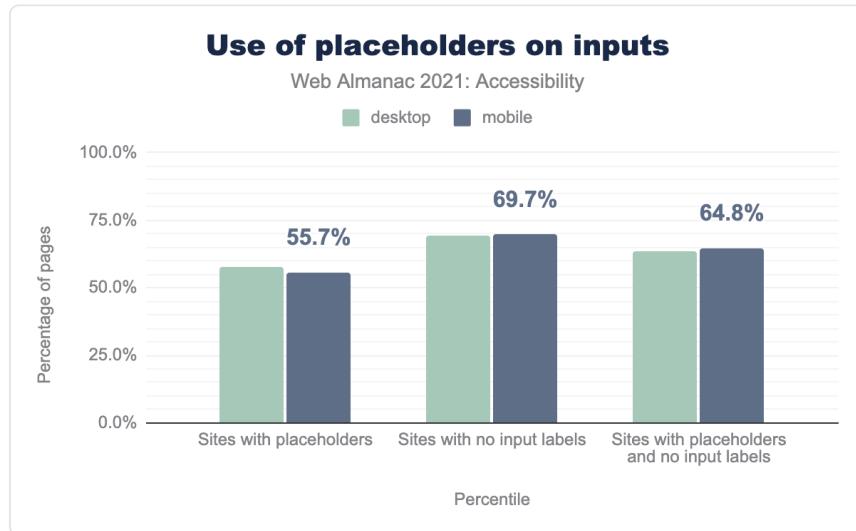


Figure 9.15. Use of placeholders on inputs.

The improper use of a placeholder as a replacement for the `<label>` element is surprisingly prevalent. Roughly 58% of desktop and mobile websites in this year's survey used the `placeholder` attribute. Of those sites, nearly 65% of them included the `placeholder` attribute and failed to include a programmatically associated `<label>` element.

There are many accessibility issues that placeholder text can present<sup>351</sup>. For example, because it disappears when the user begins to type, people with cognitive disabilities can be disoriented and lose context for the purpose of the form element.

The HTML5 specification<sup>352</sup> clearly states, “The placeholder attribute should not be used as an alternative to a label.”

The W3C’s Placeholder Research<sup>353</sup> lists 26 different articles that advise against the flawed

351. <https://www.smashingmagazine.com/2018/06/placeholder-attribute/>

352. <https://html.spec.whatwg.org/#the-placeholder-attribute>

353. [https://www.w3.org/WAI/GL/low-vision-a11y-tf/wiki/Placeholder\\_Research](https://www.w3.org/WAI/GL/low-vision-a11y-tf/wiki/Placeholder_Research)

design approach of using a placeholder instead of the semantically correct `<label>` element. It goes on to say:

*Use of the placeholder attribute as a replacement for a label can reduce the accessibility and usability of the control for a range of users including older users and users with cognitive, mobility, fine motor skill or vision impairments.*

— The W3C's Placeholder Research<sup>354</sup>

## Requiring information

When web developers gather input from their end users, they need a clear way to indicate what information is optional, and what information is required to proceed. For example, a shipping address is optional if the end user is buying something online that they can download. However, the method of payment is most likely required in order to complete the sale.

Before HTML5 introduced the `required` attribute for `<input>` fields in 2014, web developers were forced to solve this problem on an ad hoc, case-by-case basis. A common convention is to put an asterisk (\*) in the label for required input fields. This is purely a visual, stylistic convention—labels with asterisks don't enforce any kind of field validation. Additionally, screen readers typically announce this character as "star" unless it is explicitly hidden from assistive technology, which can be confusing.

There are two attributes that can be used to communicate the required state of a form field to assistive technology. The `required` attribute will be announced by most screen readers and actually prevents form submission when a required field has not been properly filled out. The `aria-required` attribute can be used to indicate required fields to assistive technology, but does not come with any associated behavior that would interfere with form submission.

---

<sup>354</sup>. [https://www.w3.org/WAI/GL/low-vision-a11y-tf/wiki/Placeholder\\_Research](https://www.w3.org/WAI/GL/low-vision-a11y-tf/wiki/Placeholder_Research)



Figure 9.16. How required inputs are specified

We found that 21% of desktop websites had form elements that have either an asterisk (\*) in their label, the `required` attribute or the `aria-required` attribute or some combination of these techniques. Two-thirds of these form elements used the `required` attribute. About a third of all required inputs used the `aria-required` attribute. Roughly 22% had an asterisk in their label.

## Media on the web

Accessibility plays an increasingly important role in all media consumption on the web. For people who are deaf or hard of hearing, captions provide access to video. For people who are blind or have vision impairments, audio descriptions can describe a scene. Without removing the barriers to access to media content, we are excluding people from the majority of what gets visited on the web.

According to this Streaming Media study<sup>355</sup>, “by 2022, video viewing will account for 82% of all internet traffic”. Whenever you use media in your web content—images, audio, or video—you must ensure it is accessible to all.

## Overview of text alternatives

Every HTML media element allows you to provide text alternatives, but not every author takes advantage of this accessibility capability.

The `<img>` element for displaying pictures was introduced in the HTML 2.0 specification in 1995. The `alt` attribute—introduced at the same time—provides a clear mechanism for the web developer to provide a text alternative for the image.

This alternative description of the image is used by screen readers to describe the image for someone who can't see the image. It is also used to describe the image to everyone if the image cannot be downloaded or displayed. One type of “user” who can't see the image is a search engine—good `alt` text plays an important role in Search Engine Optimization (SEO), so that web pages that show the image can be discovered by text searches.

The HTML5 specification introduced the `<video>` and `<audio>` elements in 2014 to provide a standards-based way to incorporate rich media in your website that didn't require a third-party browser plugin. Both the `<video>` and `<audio>` elements allow a `<track>` element to be included, so that closed captions, subtitles, and audio descriptions can provide alternate, text-based ways to enjoy the rich media.

These tracks provide the same SEO benefits as `alt` text does for images, although in 2021, less than 1% of the websites surveyed provided `<track>` elements.

## Images

The `alt` attribute allows web authors to provide a text alternative for the visual information communicated in an image. A screen reader can convey its visual meaning through audio by announcing the image's alternative text. Additionally, if images are unable to load, the alternative text for a description will be displayed.

Images need to be described appropriately, in some cases short descriptions are helpful, and in other cases a longer description is needed to capture the meaning or intent of the image.

The 2021 Lighthouse audit data shows that 57% of sites pass the test for images with `alt` text, a small increase from 54% the year before. This test looks for the presence of at least one

<sup>355.</sup> <https://www.streamingmedia.com/Articles/ReadArticle.aspx?ArticleID=144177>

of the `alt`, `aria-label` or `aria-labelledby` attributes on `img` elements. In most cases using the `alt` attribute is the best choice.



Figure 9.17. Pages containing an `alt` attribute with a file extension.

Automated checks for the presence of alternative text usually do not assess the quality of this text. One unhelpful pattern is describing the image with the file extension name. We found that 7.1% of desktop sites (with at least one instance of the `alt` attribute) had a file extension in the value of at least one `img` element's `alt` attribute, compared to 7.3% the previous year.

## Most common file extensions in alt text

Web Almanac 2021: Accessibility



Figure 9.18. Most common file extensions in alt text.

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.



Figure 9.19. `alt` attribute lengths.

We found that 27% of `alt` text attributes were empty. In an ideal world this would indicate that the associated images are decorative<sup>356</sup>, and should not be described by assistive technologies. However, the majority of images add value to an interface and as such, should be described. We found that 15% have 10 or fewer characters, which would be a strangely short description for most images, indicating that information parity has not been achieved.

## Audio

`<track>` provides a way for a text equivalent to be provided for audio in `<audio>` and `<video>` elements. This allows people with permanent or temporary hearing loss to be able to understand audio content.

**0.02%**

Figure 9.20. Desktop websites with an `<audio>` element have at least one accompanying `<track>` element

`<track>` loads one or more WebVTT files, which allows text content to be synchronized with

356. <https://www.w3.org/WAI/tutorials/images/decorative/#--text-for-example%2C%20the%20information%20provided,technologies%2C%20such%20as%20screen%20readers>.

the audio it is describing. We found 0.02% of all pages on desktop and 0.05% of all pages on mobile with a detectable `<audio>` element had at least one accompanying `<track>` element.

These data points do not include audio embedded via an `<iframe>` element, which is common for content like podcasts that use a third-party service to host and list recordings.

## Video

The `<video>` element was only present on roughly 5% of the websites included in the 2021 Web Almanac.



Figure 9.21. Desktop websites with an `<video>` element have at least one accompanying `<track>` element

Similar to the results of the `<audio>` survey, the `<track>` element was included with a corresponding `<video>` element less than 1% of the time—0.5% for desktop sites, and 0.6% for mobile sites. In actual numbers, only 2,836 desktop sites out of 6.3 million included a `<track>` element where a `<video>` element was present. Only 2,502 mobile sites out of 7.5 million made their videos accessible by including a corresponding `<track>` element with content loaded via the `<video>` element.

Much like the `<audio>` element, this figure may not account for video content loaded by a third party `<iframe>`, such as an embedded YouTube video. It should also be noted that most popular third-party audio and video embedding services include the ability to add synchronized text equivalents.

## Supporting assistive technology with ARIA

Accessible Rich Internet Applications<sup>357</sup>—or ARIA—is a suite of web standards that was first published by the Web Accessibility Initiative in 2014. ARIA provides a set of attributes we can add to HTML markup to enhance the experience for users of assistive technology.

There are many nuances and complexities to the use of ARIA, as well as varying degrees of assistive technology support. As a general rule, it should be used sparingly, and never in

<sup>357</sup>. <https://www.w3.org/WAI/standards-guidelines/aria/>

instances when there is an equivalent native HTML solution that could be leveraged. While ARIA can provide helpful information to assistive technology, it comes with no associated behavior such as keyboard operability.

The 5 rules of ARIA<sup>358</sup> describe some helpful guiding principles for ARIA usage. In September of 2021, a W3 working group published ARIA in HTML<sup>359</sup>, a proposed specification with very detailed information about how and when ARIA can be used.

## ARIA roles

When assistive technology encounters an element, the element's role communicates information about how someone might interact with its content. For example, an `<a>` element will expose a link role to assistive technology, which typically conveys that the element will navigate somewhere when activated.

HTML5 introduced many new native elements, all which have implicit semantics<sup>360</sup>, including roles. For example, the `<nav>` element has an implicit `role="navigation"` and does not need to have this role added explicitly via ARIA in order to convey its purpose information to assistive technology.

ARIA can be used to explicitly add roles to content that does not have a fitting native HTML role. For example, when creating a `tablist` widget, a `tablist` role can be assigned to the container element since there is no native HTML equivalent.

358. <https://www.w3.org/TR/using-aria/>  
359. <https://www.w3.org/TR/2021/PR-html-aria-20210930/#priv-sec>  
360. [https://www.w3.org/TR/wai-aria-1.1/#implicit\\_semantics](https://www.w3.org/TR/wai-aria-1.1/#implicit_semantics)

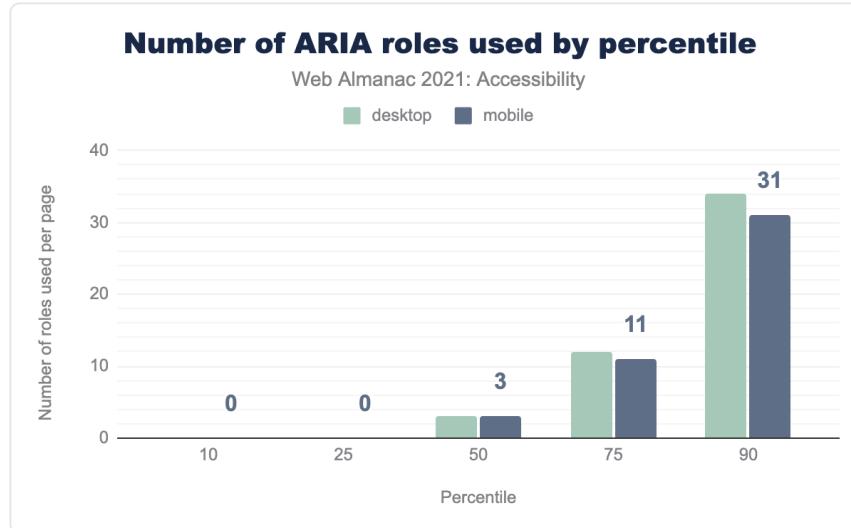


Figure 9.22. Number of ARIA roles used by percentile.

Currently 69% (up from 65% in 2020) of desktop pages have at least one instance of an ARIA role attribute. The median site has 3 instances (up from 2 in 2020) of the role attribute. The most commonly used roles are listed below.



Figure 9.23. Top 10 most common ARIA roles.

## Just use a button!

One of the most common misuses of ARIA roles is adding a button role to non-interactive elements such as `<div>`s and `<span>`s, or to `<a>` elements. A native HTML `<button>` element comes with an implicit button role and the expected keyboard operability and behavior and should be the first approach before reaching for ARIA.

We found that 29% (up from 25% in 2020) of desktop sites and 29% of mobile sites (up from 25% in 2020) had homepages with at least one element with an explicitly assigned `role="button"`. This suggests that close to a third 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.

If non-interactive elements such as `<div>`s and `<span>`s have been assigned a button role, there is a significant chance that the expected keyboard focus order and operability will not be

applied, which would result in WCAG 2.1.1 Keyboard<sup>361</sup> and 2.4.3 Focus order problems<sup>362</sup>. In addition, Windows High Contrast Mode will not honor ARIA<sup>363</sup>, so elements that are not native HTML button elements may not appear to be interactable in this mode. We found that 11% of desktop and mobile sites have either a `<div>` or a `<span>` with an explicit `button` role.

When a button role is applied to an `<a>` element, it overrides the implicit link role that anchor elements come with. This can lead to a confusing user experience because the expected behavior for a button would be to trigger an in-page action, whereas a link would typically navigate somewhere. There would also be a violation WCAG 2.1.1, Keyboard<sup>364</sup> if the correct keyboard behavior has not been implemented (links are not activated with the space key, whereas buttons are). Additionally, when a button role is announced by a screen reader without the expected corresponding behavior, it can create a confusing and disorienting experience for an assistive technology user.

# 18.6%

*Figure 9.24. Desktop websites have at least one link with a button role*

We found that 19% of desktop pages (up from 16% in 2020) and 18% (up from 15% in 2020) of mobile pages contained at least one anchor element with `role="button"`. A native `<button>` element would be a better choice, per the first rule of ARIA<sup>365</sup>.

This act of adding ARIA roles, or a “role-up”<sup>366</sup>, is usually less ideal than using the correct native HTML element. Again, 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, as it comes with the expected semantics and behavior.

## Using presentation role

When an element has `role="presentation"` declared on it, its semantics are stripped away, as well as any of its child elements. For example, declaring `role="presentation"` on a parent table or list element will cascade the role to any child elements. This will also strip the semantics.

Removing an element’s semantics means that it is no longer that element in terms of its behavior or how it is understood by assistive technology, leaving only its visual appearance. For

361. <https://www.w3.org/TR/UNDERSTANDING-WCAG20/keyboard-operation-keyboard-operable.html>

362. <https://www.w3.org/TR/UNDERSTANDING-WCAG20/navigation-mechanisms-focus-order.html>

363. <https://ericwbailydesign/writing/truths-about-digital-accessibility/#windows-high-contrast-mode-ignores-aria>

364. <https://www.w3.org/TR/UNDERSTANDING-WCAG20/keyboard-operation-keyboard-operable.html>

365. <https://www.w3.org/TR/using-aria/#rule1>

366. <https://adrianroselli.com/2020/02/role-up.html>

example, a list with a `role="presentation"` will no longer communicate any information to a screen reader about the list structure. We found that 22% of desktop pages and 21% of mobile pages have at least one element with `role="presentation"`. There are very few use cases where this is particularly helpful for assistive technology users, so use this role sparingly and thoughtfully.

## Labelling and describing elements with ARIA

Parallel to the DOM there is a similar browser structure called the accessibility tree<sup>367</sup>. It contains information about HTML elements including accessible names, descriptions, roles and states. This information is conveyed to assistive technology through accessibility APIs.

The 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 or targeted by assistive technology.

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). There is a specificity ranking that happens to determine which value is assigned to the accessible name if there are multiple potential sources.

For more information about accessible names visit Léonie Watson<sup>368</sup>'s article, What is an accessible name?<sup>369</sup>

We can also use ARIA to provide accessible names for elements. There are two ARIA attributes that accomplish this, `aria-label` and `aria-labelledby`. Either of these attributes will “win” the accessible name computation and override the natively derived accessible name. It is important to use these two attributes 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 your users will expect. When using ARIA to name an element, it is important to ensure that the WCAG 2.5.3, Label in Name<sup>370</sup> criterion has not been violated, which expects visible labels to be at least a part of its accessible name.

<sup>367</sup>. [https://developer.mozilla.org/docs/Glossary/Accessibility\\_tree](https://developer.mozilla.org/docs/Glossary/Accessibility_tree)

<sup>368</sup>. <https://twitter.com/LeonieWatson>

<sup>369</sup>. <https://developer.paciellogroup.com/blog/2017/04/what-is-an-accessible-name/>

<sup>370</sup>. <https://www.w3.org/WAI/WCAG21/Understanding/label-in-name.html>



Figure 9.25. Top 10 ARIA attributes.

The `aria-label` attribute allows a developer to provide a string value, and this will be used for the accessible name for the element. It is worth noting that voice to text users may have difficulty targeting controls that are named without visible text as a reference. People with cognitive disabilities often benefit from visible text as well. An invisible accessible name is better than no accessible name, however, in most cases, a visible label should either supply the accessible name or at a minimum be contained within an element's accessible name.

We found that 53% of desktop pages (up from 40% in 2020) and 52% of mobile home pages (up from 39% in 2020) had at least one element with the `aria-label` attribute, making it the most popular ARIA attribute for providing accessible names, with a very large increase in usage in 1 year. This could be a positive indication that more elements that previously were lacking an accessible name now have one. However, it could also signify an increase in elements having no visible label, which could negatively impact people with cognitive disabilities and voice to text users.

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 21% of desktop pages (up from 18% in 2020) and 20% of mobile pages (up from 16% in 2020) had at

least one element with the `aria-labelledby` attribute.

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 13% of desktop pages and 12% of mobile pages had at least one element with the `aria-describedby` attribute.

*Fun fact! We found 1,886 websites with the attribute `aria-lavel`, which is a misspelling of the `aria-label` attribute! Be sure to run those automated checks to pick up these easily avoidable errors.*

## Where do buttons get their accessible names from?

Buttons typically get their accessible names from their content or an ARIA attribute. Per the first rule of ARIA<sup>371</sup>, if an element can derive its accessible name without the use of ARIA, this is preferable. Therefore a `<button>` should get its accessible name from its text content rather than an ARIA attribute if possible.

There is a common implementation where text content is not used to supply the accessible name because the button is a graphical control using an image or icon. This can be problematic for voice to text users who need to target the control without visible text and should not be used if visible text is an option.

371. <https://www.w3.org/TR/using-aria/#rule1>

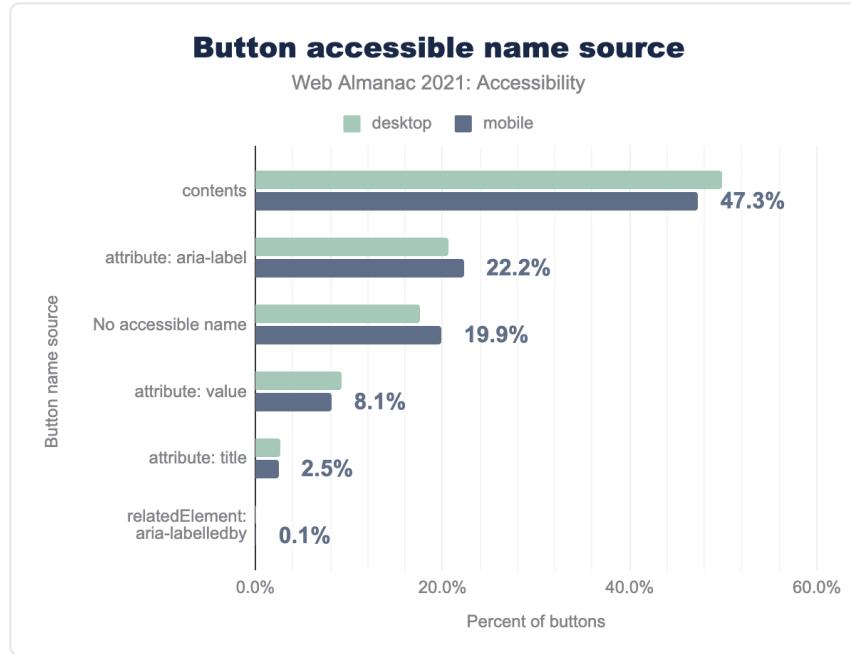


Figure 9.26. Button accessible name source.

We found that 57% of buttons on both desktop and mobile sites get their accessible name from content. We also found that 29% of buttons on desktop sites and 27% of buttons on mobile sites get their accessible names from the `aria-label` attribute.

## Hiding content

There are several ways to ensure that assistive technology will not discover content. We can leverage CSS `display: none;` 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"`. Note that unlike `display: none;` a declaration of `aria-hidden="true"` will not visibly remove an element and its children.

**53.8%**

Figure 9.27. Desktop websites have at least one instance of the `aria-hidden` attribute

We found that 54% of desktop pages (up from 48% in 2020) and 53% of mobile pages (up from

49% in 2020) had at least one instance of an element with the `aria-hidden` attribute.

These techniques are most helpful when something in the visual interface is redundant or unhelpful to assistive technology users. Hiding content from assistive technology should never be used 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 hidden UI for everyone. Hide/show widgets should be making use of the `aria-expanded` attribute to indicate to assistive technology that something can be revealed when the control is activated and hidden when activated again. We found that 26% of desktop pages (up from 21% in 2020) and 25% of mobile pages (up from 21% in 2020) had at least one element with the `aria-expanded` attribute.

## Screen reader-only text

A common technique that developers employ to supply additional information for screen reader users is to use CSS to visually hide a passage of text but make it discoverable by a screen reader. Since `display: none;` prevents content from being present in the accessibility tree, there is a common pattern involving a specific set of declarations of CSS code.



*Figure 9.28. Desktop websites with a `sr-only` or `visually-hidden` class*

The most common CSS class names for this code snippet<sup>372</sup> (both by convention and throughout libraries like Bootstrap) are `sr-only` and `visually-hidden`. We found that 14% of desktop pages and 13% of mobile pages had one or both of these CSS class names. It is worth noting that there are screen reader users who have some vision, therefore over-reliance on visually hidden text could be confusing for some.

## Dynamically-rendered content

The presence of new or updated content in the DOM sometimes needs to be communicated to screen readers. Some thought needs to be put into which updates need to be conveyed to avoid frustration. For example, form validation errors need to be conveyed whereas a lazy-loaded image may not. Updates to the DOM also need to be done in a way that is not disruptive.

---

<sup>372.</sup> <https://css-tricks.com/inclusively-hidden/>

ARIA live regions allow us to listen for changes in the DOM, such that the updated content can be announced by a screen reader. We found that 21% of desktop pages (up from 17% in 2020) and 20% of mobile pages (up from 16% in 2020) have live regions. For more information about live region variants and usage check out the MDN live region documentation<sup>373</sup> or play with this live demo by Deque<sup>374</sup>.

## Accessibility overlays

Accessibility overlays, sometimes referred to as accessibility plugins or overlay widgets, are digital products that are marketed as tools to easily solve a website's accessibility issues. The Overlay Fact Sheet<sup>375</sup> defines them as “a broad term for technologies that aim to improve the accessibility of a website. They apply third-party source code (typically JavaScript) to automate improvements to the front-end code of the website.”

Many of these products have deceptive marketing materials suggesting that one line of code can make websites accessible, or at least legally compliant from an accessibility standpoint.

For example, accessiBe<sup>376</sup>, one of the most aggressive products in this space, explains their process as being able to make sites accessible and compliant within 48 hours by simply pasting their JavaScript installation code into production code.

Unfortunately, web accessibility is simply not possible to achieve with an out of the box solution like this. If it were, we would likely not see the sobering statistics throughout this chapter.

---

373. [https://developer.mozilla.org/docs/Web/Accessibility/ARIA/ARIA\\_Live\\_Regions](https://developer.mozilla.org/docs/Web/Accessibility/ARIA/ARIA_Live_Regions)

374. <https://dequeuniversity.com/library/aria/liveregion-playground>

375. <https://overlayfactsheet.com/#what-is-a-web-accessibility-overlay>

376. <https://en.wikipedia.org/wiki/AccessiBe>



Figure 9.29. Pages using accessibility apps.

We found that 0.96% of desktop websites—or well over 60,000—use one of these accessibility overlays. It is worth noting that we have queried for a list of well-known products in this space. However, this list is not exhaustive, so this metric is likely higher in reality.



Figure 9.30. Accessibility app usage by rank.

When considering domain rank, the top 1,000 websites have a lower percentage –0.1%— of overlay usage. However, considering the reach of these top-ranking sites, the potential impact of even one website with this much traffic using an overlay is very substantial.



Figure 9.31. Pages using accessibility apps by rank.

## The consequences of overlays

These tools often interfere with assistive technologies and actually make websites less accessible for many, as is explored by a Vice article aptly titled “People with Disabilities Say This AI Tool is Making the Web Worse for Them”<sup>377</sup>. There is even an open-source extension called accessiByeBye<sup>378</sup> that was specifically developed to block overlays so that assistive technology users are not disrupted in their use of websites use a third-party overlay product.

As civil rights lawyer Haben Girma<sup>379</sup> explains in this video about accessibility overlays<sup>380</sup>, “AI is a tool and right now it is extremely limited in what it can do for accessibility”. She goes on to explain how auto-generated captions of her name misinterpreted “Haben Girma” as “happen grandma” and how this type of miscommunicated information can impact deaf users.

There have been tensions between some of these overlay companies and the disabled communities they purport to serve. For example, The National Federation of the Blind banned

377. <https://www.vice.com/en/article/m7az74/people-with-disabilities-say-this-ai-tool-is-making-the-web-worse-for-them>

378. <https://www.accessibyebye.org/>

379. <https://twitter.com/HabenGirma>

380. <https://www.youtube.com/watch?v=R1ZZ1Sp-u4U>

accessiBe from their national convention<sup>381</sup> and released a this statement about the harm caused by the company<sup>382</sup>.

*It seems that accessiBe fails to acknowledge that blind experts and regular screen reader users know what is accessible and what is not. The nation's blind will not be placated, bullied, or bought off.*

— National Federation for the Blind<sup>383</sup>

## Privacy concerns

Some of these tools have techniques for detecting the use of assistive technologies. This means that personal data is potentially collected about a person's disabilities without their consent.

From the Overlay Fact Sheet<sup>384</sup>:

*Some overlays have been found to persist users' settings across sites which use the same overlay. This is done by setting a cookie on the user's computer.*

*When the user enables a setting for an overlay feature on one site, the overlay will automatically turn on that feature on other sites... the big privacy problem is that the user never opted in to be tracked and there's also no ability to opt-out. Due to this lack of an opt-out (other than explicitly turning off that setting) this creates General Data Protection Regulation (GDPR) and California Consumer Privacy Act (CCPA) risk for the overlay customer.*

— Overlay Fact Sheet<sup>385</sup>

This article by Léonie Watson<sup>386</sup> explores the privacy concerns of this type of data tracking in accessibility overlays.

## Overlays and lawsuits

These widgets have been named as part of many accessibility lawsuits against companies who

<sup>381</sup>. <https://www.forbes.com/sites/gusalexiou/2021/06/26/largest-us-blind-advocacy-group-bans-web-accessibility-overlay-giant-accessibe/?sh=16621ec55a15>

<sup>382</sup>. <https://nfb.org/about-us/press-room/national-convention-sponsorship-statement-regarding-accessible>

<sup>383</sup>. <https://nfb.org/about-us/press-room/national-convention-sponsorship-statement-regarding-accessible>

<sup>384</sup>. <https://overlayfactsheet.com/#privacy>

<sup>385</sup>. <https://overlayfactsheet.com/>

<sup>386</sup>. <https://tink.uk/accessibe-and-data-protection/>

use them. According to the UsableNet's 2020 report on Digital Accessibility Lawsuits<sup>387</sup>, "Over 250 companies sued had invested in accessibility widgets or overlays". Accessibility expert Sherri Byrne-Haber cites<sup>388</sup>, "Ten percent of accessibility lawsuits filed at the end of 2020 were against companies who have installed plugins, overlays, or widgets, thinking they would make them bulletproof to ADA litigation". It's worth noting that accessibility laws are not limited to the Americans with Disabilities Act, there are countries all over the world with laws pointing to the WCAG<sup>389</sup>.

For more information about the legal implications of using these overlays, refer to Lainey Feingold<sup>390</sup>'s article Honor the ADA: Avoid Web Accessibility Quick-Fix Overlays<sup>391</sup> and Adrian Roselli's article #accessiBe Will Get You Sued<sup>392</sup>.

## Why do some companies use overlays?

Fundamentally, and fueled by ableism<sup>393</sup>, overlays position themselves as solving a problem that most organizations struggle with. The data is clear throughout this chapter—the internet is largely inaccessible.

These products take advantage of gaps in organizational accessibility knowledge. Their framing of the problem space aims to help avoid lawsuits by automating solutions, rather than meaningfully removing barriers to access for people with disabilities. The reason these lawsuits happen is that there are real Civil Rights violations when people's right to access online is infringed upon. For example, an AI tool supplying a poor accessible description for an image might pass the checks of an automated tool, but this does not remove the barrier for a blind person or offer information parity.

Organizations can be swayed by the deceptive marketing of some of these overlay companies promising to make their products accessible and fully compliant with one line of code and a few dollars a month. The unfortunate reality is that these tools introduce new barriers for people with disabilities and can open the organization up to unforeseen legal issues.

There is no quick fix—the onus is on organizations and digital practitioners to prioritize actually fixing the accessibility problems in their web content. A common saying amongst the disabled community is, "nothing about us without us". Overlays have been created without much involvement from the disabled community, and some of these companies have further alienated people with disabilities who have spoken out about this<sup>394</sup>. These products cannot achieve equal access to the web for people with disabilities.

387. <https://info.usablenet.com/2020-report-on-digital-accessibility-lawsuits>

388. <https://sheribynehaber.com/technology-doesnt-make-accessibility-hard-people-who-dont-care-do/>

389. <https://www.3playmedia.com/blog/countries-that-have-adopted-wcag-standards-map/>

390. <https://twitter.com/LFLegal>

391. <https://www.lflegal.com/2020/08/quick-fix/>

392. <https://adrianroselli.com/2020/06/accessibe-will-get-you-sued.html>

393. <https://www.forbes.com/sites/andrewpulrang/2020/10/25/words-matter-and-its-time-to-explore-the-meaning-of-ableism/?sh=7ab349837162>

394. <https://www.nbcnews.com/tech/innovation/blind-people-advocates-slam-company-claiming-make-websites-ada-compliant-n1266720>

## Additional resources about overlays

- Connor Scott-Gardener's experience using an overlay<sup>395</sup>
- Case study of an ADA lawsuit involving an overlay<sup>396</sup>
- The A11y Project - Should I Use an Accessibility Overlay?<sup>397</sup>
- There's no such thing as fully automated web accessibility<sup>398</sup>
- Why Automated Tools Alone Can't Make Your Website Accessible and Legally Compliant<sup>399</sup>
- Should I Use an Accessibility Overlay?<sup>400</sup>

## Conclusion

As accessibility advocate Billy Gregory once said<sup>401</sup>, “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.

The entire product team and organization have to prioritize accessibility as part of their accountabilities in order to succeed, all the way up to the C-suite. Accessibility work needs to shift left in the product cycle<sup>402</sup>, meaning it needs to be baked into the research, ideation and design stages before it is developed. And most importantly, people with disabilities need to be included in this process.

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.

As an industry it is time that we acknowledge the story told by the numbers in this chapter; we are failing people with disabilities. The numbers from 2021 have not moved substantially from 2020. We need to do better, and this has to come from a combination of top-down leadership and investment (including the ongoing participation from browsers) and bottom-up effort to

395. <https://catchtheswords.com/do-automated-solutions-like-accessible-make-the-web-more-accessible/>

396. <https://uxdesign.cc/important-settlement-in-an-ada-lawsuit-involving-an-accessibility-overlay-748a82850249>

397. <https://www.a11yproject.com/posts/2022-03-08-should-i-use-an-accessibility-overlay/>

398. <https://uxdesign.cc/theres-no-such-thing-as-fully-automated-web-accessibility-260d6f4632a8>

399. <https://www.forbes.com/sites/gusalexiou/2021/10/28/why-automated-tools-alone-can-t-make-your-website-accessible-and-legally-compliant/?sh=2e538b62364e>

400. <https://shoulduseanaccessibilityoverlay.com/>

401. <https://twitter.com/thebillygregory/status/552466012713783297?s=20>

402. <https://feather.ca/shift-left/>

push our practices forward and advocate for the needs, safety and inclusion of people with disabilities using the web.

## Authors

---



### Alex Tait

Twitter: @at\_fresh\_dev GitHub: alextait1 Website: <https://atfreshsolutions.com>

Alex Tait is an accessibility specialist 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.



### Scott Davis

Github: scottdavis99

Scott Davis is an author and Digital Accessibility Advocate with Thoughtworks<sup>403</sup>, where he focuses on leading-edge / innovative / emerging / non-traditional aspects of web development. “Digital Accessibility is so much more than a compliance checkbox; Accessibility is a springboard for innovation.”



### Olu Niyi-Awosusi

Twitter: @oluoluoxenfree GitHub: oluoluoxenfree Website: <https://olu.online/>

Olu Niyi-Awosusi is a JavaScript engineer at Oddbird<sup>404</sup> who loves lists, learning new things, Bee and Puppycat, social justice, accessibility<sup>405</sup> and trying harder every day.

---

403. <https://www.thoughtworks.com/>

404. <https://www.oddbird.net/>

405. <https://alistapart.com/article/building-the-woke-web/>



## Gary Wilhelm

👤 gwilhelm

Gary Wilhelm is the Digital Solutions Manager for the Division of Finance and Operations at UNC-Chapel Hill<sup>406</sup>, which is a fancy way of saying that he works on websites and develops web applications. He started working to make his websites accessible in 2013 by studying specifications and has been interested in accessibility ever since, including spending large amounts of time learning about PDF accessibility through remediating several thousand PDF documents. In his spare time, he likes to travel, do yard work, run, watch sports, pester his wife and two teenagers, and help his dog look for squirrels and rabbits.



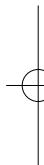
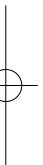
## Katriel Paige

👤 kachiden 🌐 <https://www.flowerstorm.tech/>

Kit Paige is an accessibility engineer and cat enthusiast who's long and winding path through tech has included QA, UX, frontend development, a love hate relationship with CSS, and immeasurable coffee.

---

406. <https://www.unc.edu/>



# Part II Chapter 10

# Performance



**Written by Sia Karamalegos**

Reviewed by Rick Viscomi, Kevin Farrugia, Estelle Weyl, Ziemek Bućko, Julia Yang, Fili Wiese, Barry Pollard, Samar Panda, and Edmond W. W. Chan

Analyzed by Sia Karamalegos, Rick Viscomi, and Nitin Pasumarthi

Edited by Julia Yang

## Introduction

Performance is important for user experience. Slow-to-load and slow-to-respond websites frustrate users and cause lost conversions. This is the first year that the Core Web Vitals<sup>407</sup> have contributed to Google search rankings<sup>408</sup>. As such, we've seen greater interest in improving website performance which is great news for users.

*What are our top takeaways from this year's report?* First, we still have a long way to go in providing a good user experience. For example, faster networks and devices have not yet reached the point where we can ignore how much JavaScript we deliver to a site; and, we may never get there. Second, sometimes we misuse new features for performance, resulting in poorer performance. Third, we need better metrics for measuring interactivity, and those are on the way. And fourth, CMS- and framework-level work on performance can significantly

407. <https://web.dev/vitals/>

408. <https://developers.google.com/search/blog/2020/11/timing-for-page-experience>

impact user experience for the top 10M websites.

*What's new this year?* We're excited to share performance data by traffic ranking for the first time. We also have all the core performance metrics from previous years. Finally, we added a deeper dive into the Largest Contentful Paint (LCP) element.

## Notes on Methodology

One thing that makes the performance chapter different from the others is that we rely heavily on the Chrome User Experience Report<sup>409</sup> (CrUX) for our analyses. Why? If our number one priority is user experience, then the best way to measure performance is with real user data (real user metrics, or RUM for short).

*The Chrome User Experience Report provides user experience metrics for how real-world Chrome users experience popular destinations on the web.*

— *Chrome User Experience Report*<sup>410</sup>

CrUX data only provides high-level field/RUM metrics and only for the Chrome browser. Additionally, CrUX reports data by origin, or website, instead of by page.

We supplement our CrUX RUM data with lab data from WebPageTest in HTTP Archive. WebPageTest includes very detailed information about each page, including the full Lighthouse report. Note that WebPageTest measures performance in locations across the U.S. The performance data in CrUX is global since it represents real user page loads.

When comparing performance year-over-year, keep in mind that:

- The Cumulative Layout Shift (CLS) calculation has changed<sup>411</sup> since 2020.
- The First Contentful Paint (FCP) thresholds ("good", "needs improvement", and "poor") have changed<sup>412</sup> since 2020.
- Last year's report was based on August 2020 data, and this year's report was based on the July 2021 run.

Read the full methodology for the Web Almanac to learn more.

409. <https://developers.google.com/web/tools/chrome-user-experience-report>

410. <https://developers.google.com/web/tools/chrome-user-experience-report>

411. <https://web.dev/cls-web-tooling/>

412. <https://web.dev/cls-web-tooling/#additional-updates>

## High-Level Performance: Core Web Vitals

Before we dive into the individual metrics, let's take a look at combined performance for Core Web Vitals<sup>413</sup> (CWV). Core Web Vitals (LCP, CLS, FID) are a set of performance metrics focused on user experience. They focus on loading, interactivity, and visual stability.

Web performance is notorious for an alphabet soup of metrics, but the community is coalescing on this framework.

This section focuses on websites that reached the “good” threshold on all three CWV metrics to understand how the web is performing at a high level. In the Analysis by Metric section, we’ll cover the same charts by each metric in detail, plus more metrics not in the CWV.

### By Device



Figure 10.1. Good Core Web Vitals by Device from 2020 to 2021

**Note:** As the CLS calculation changed since last year, this is not an apples-to-apples comparison.

Core Web Vitals for websites in the Chrome User Experience Report improved year-over-year. But, a good part of this improvement could be due to a change in the CLS calculation, not necessarily to a performance improvement in CLS. The resulting CLS “improvement” was 8 points on desktop (2 for mobile). LCP improved by 7 points for desktop (2 for mobile). FID was

413. <https://web.dev/vitals/>

already at 100% for desktop for both years and improved by 10 points on mobile.

As in previous years, performance was better on desktop machines than mobile devices. This is why it's crucial to test your site's performance on real mobile devices and to measure real user metrics (i.e., field data). Emulating mobile in developer tools is convenient in the lab (i.e., development) but not representative of real user experiences.

## By Effective Connection Type

The data by connection type in CrUX can be difficult to understand. It is not based on traffic. If a website has any experiences in a connection type, then it increases the denominator for that connection type. If the experiences were good for that website in that connection type, then it increases the numerator. Said another way, for all the websites which experienced page loads at 4G speed, 36% of those websites had good CWV:



Figure 10.2. Good CWV performance by effective connection type

Faster connections correlated with better Core Web Vitals performance. Offline performance was better presumably because of service worker caching in progressive web apps. Yet, the number of origins in the offline effective connection type category is negligible at 2,634 total (0.02%).

The top takeaway is that 3G and lower speeds correlated with significant performance degradation. Consider providing pared-down experiences for access at low connection speeds

(e.g., data saver mode<sup>414</sup>). Profile your site with devices and connections that represent your users (based on your analytics data).



*Figure 10.3. Change in effective connection type 2020-2021*

Earlier, we mentioned year-over-year improvements in LCP and FID improvements. These could be partly due to faster mobile devices and mobile networks. The chart above shows total origins accessed on 3G dropped by 2 percentage points while 4G access increased by 3 percentage points. Percent of origins is not necessarily correlated with traffic. But, I would guess if people have more access to higher speeds, then more origins would be accessed from that connection type.

Performance by connection type would be easier to understand if we could start tracking by traffic and not just origin. It would also be nice to see data for higher speeds. However, the API is currently limited<sup>415</sup> to grouping anything above 4G as 4G.

414. <https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/save-data/>

415. [https://developer.mozilla.org/docs/Glossary/Effective\\_connection\\_type](https://developer.mozilla.org/docs/Glossary/Effective_connection_type)

## By Geographic Region

### Top 30 regions for good CWV performance

Web Almanac 2021: Performance (Chrome UX Report 202107)

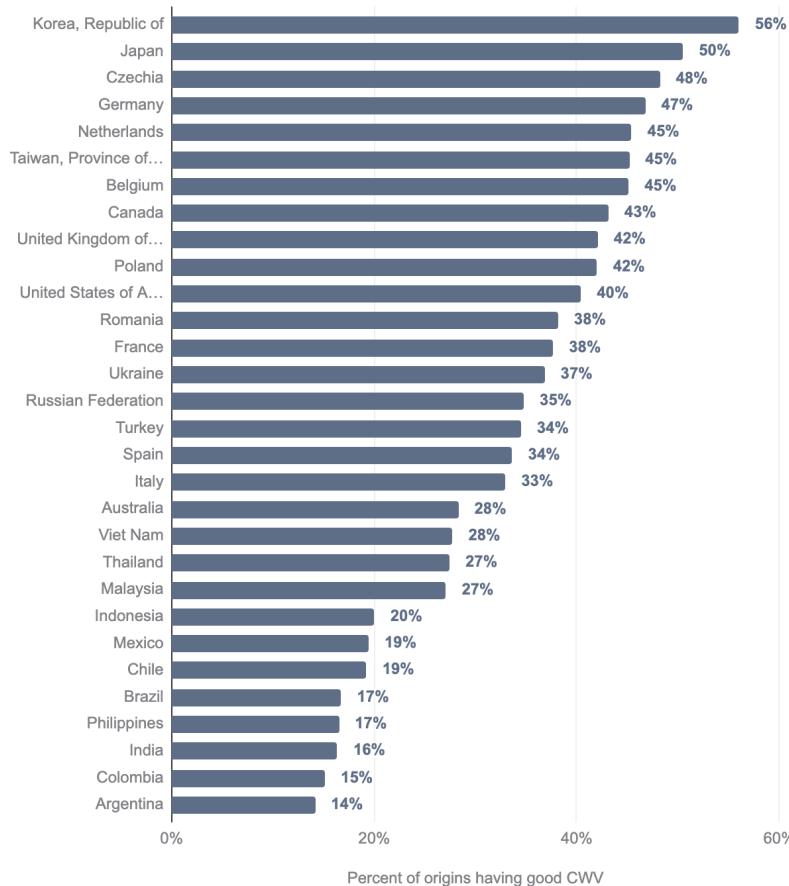


Figure 10.4. Top 30 regions for good CWV performance

Regions in parts of Asia and Europe continued to have higher performance. This may be due to higher network speeds, wealthier populations with faster devices, and closer edge-caching locations. We should understand the dataset better before drawing too many conclusions.

CrUX data is only gathered in Chrome. The percent of origins by country does not align with

relative population sizes. Reasons may include differences in browser share, in-app browsing, device share, level of access, and level of use. Keep these caveats in mind when evaluating regional-level differences and context for all CrUX analyses.

## By Rank

This year for the first time, we have ranking data! CrUX determines ranking by the number of page views per website measured in Chrome. In the charts, the categories are additive. The top 10,000 sites include the top 1,000 sites, and so forth. See the methodology for more details.



*Figure 10.5. Good CWV performance by rank*

The top 1,000 sites significantly outperformed the rest in Core Web Vitals. An interesting trough of poorer performance occurs in the middle of the chart which is due to CLS. FID was flat across all groupings. All other metrics correlated with higher performance for higher ranking.

Correlation is not causation. Yet countless companies have shown performance improvements leading to bottom-line business impacts (WPO stats<sup>416</sup>). You don't want performance to be the reason you can't achieve higher traffic and increased engagement.

416. <https://wpostats.com/>

## Analysis by Metric

In this section, we dive into each metric. For those who are less familiar, we've included links to articles that explain each metric in depth.

### Time-to-First-Byte (TTFB)

Time-to-first-byte<sup>417</sup> (TTFB) is the time between the browser requesting a page and when it receives the first byte of information from the server. It is the first metric in the chain for website loading. A poor TTFB will result in a chain reaction impacting FCP and LCP. It's why we're talking about it first.



Figure 10.6. TTFB performance by device

TTFB was faster on desktop than mobile, presumably because of faster network speeds.

Compared to last year<sup>418</sup>, TTFB marginally improved on desktop and slowed on mobile.

417. <https://web.dev/ttfb/>

418. [https://almanac.httparchive.org/en/2020/performance#fig-17\\_](https://almanac.httparchive.org/en/2020/performance#fig-17_)



Figure 10.7. TTFB performance by connection type

We have a long way to go for TTFB. 75% of our websites were in the 4G connection group and 25% in the 3G group, with the remaining ones negligible. At 4G effective speeds, only 19% of origins had “good” performance.

You may be asking yourself how TTFB can even occur with offline connections. Presumably, most of the offline sites that record and send TTFB data use service worker caching<sup>419</sup>. TTFB measures how long it takes the first byte of the response for the page to be received, even if that response is coming from the Cache Storage API or the HTTP Cache. An actual server doesn’t have to be involved. If the response requires action from the service worker, then the time it takes the service worker thread to start up and handle the response can also contribute to TTFB. But even considering service worker startup times, these sites on average receive their first byte faster than the other connection categories.

419. [https://developer.mozilla.org/docs/Web/Progressive\\_web\\_apps/Offline\\_Service\\_workers](https://developer.mozilla.org/docs/Web/Progressive_web_apps/Offline_Service_workers)



Figure 10.8. TTFB performance by rank

For rank, TTFB was faster for higher-ranking sites. One reason could be that most of these are larger companies with more resources to prioritize performance. They may focus on improving server-side performance and delivering assets through edge CDNs. Another reason could be selection bias - the top origins might be accessed more in regions with closer servers, i.e., lower latency.

One more possibility has to do with CMS adoption. The CMS Chapter shows CMS adoption by rank.



*Figure 10.9. CMS adoption by rank*

42% of pages (mobile) in the “all” group used a CMS whereas the top 1,000 sites only had 7% adoption.

Then, if we look at the top 5 CMSs by rank, we see that WordPress has the highest adoption at for 33.6% of “all” pages:

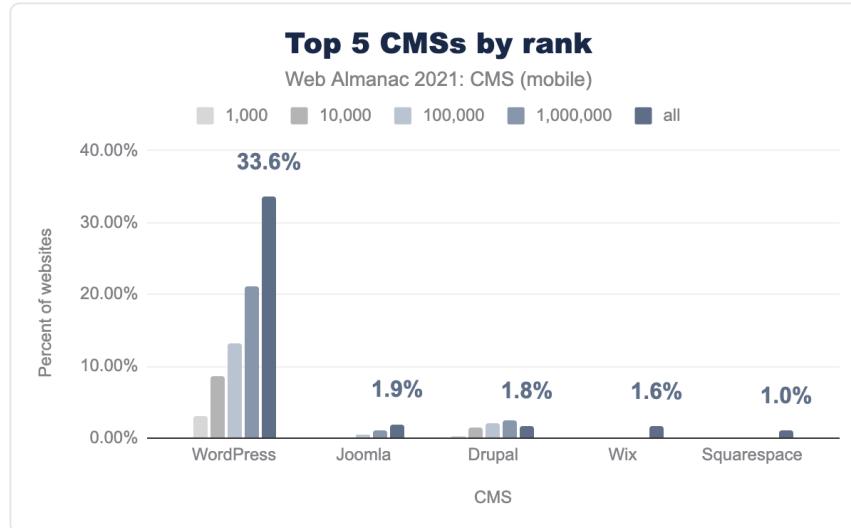
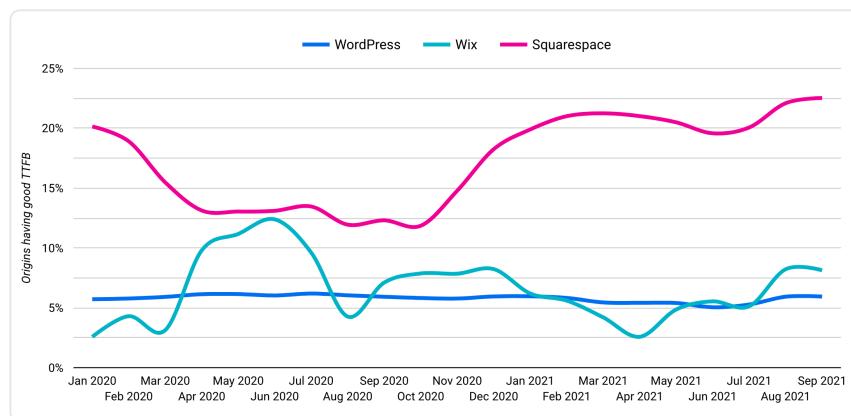


Figure 10.10. Top 5 CMSs by rank

Finally, if we look at the Core Web Vitals Technology Report<sup>420</sup>, we see how each CMS performs by metric:

Figure 10.11. Origins having good TTFB by CMS (Core Web Vitals Technology Report<sup>421</sup>)

Only 5% of origins on WordPress experienced good TTFB in July 2021. Considering WordPress's large share of the top 10M sites, its poor TTFB could be a contributor to the TTFB degradation by rank.

420. <https://datastudio.google.com/s/o6zLzTpWal>

421. <https://datastudio.google.com/s/o6zLzTpWal>

## First Contentful Paint (FCP)

First Contentful Paint (FCP)<sup>422</sup> measures the time from when a load first begins until the browser first renders any contentful part of the page (e.g, text, images, etc.).

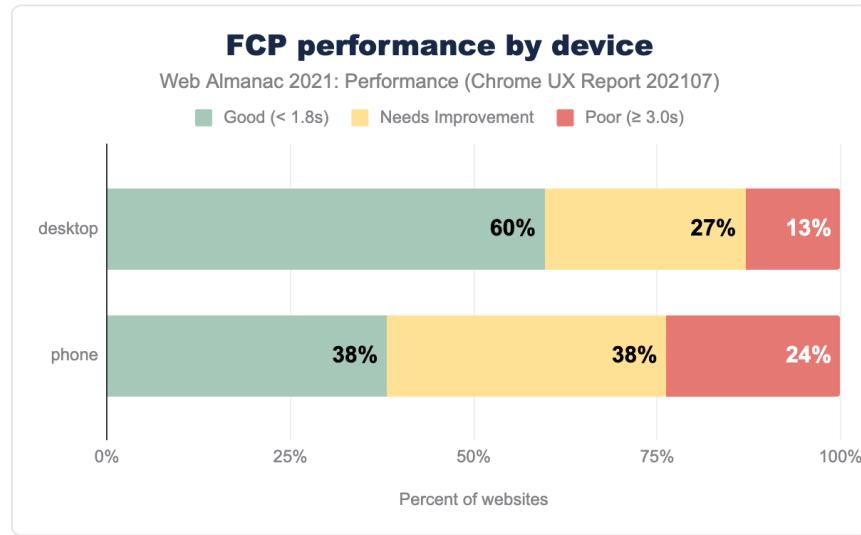


Figure 10.12. FCP performance by device

FCP was faster on desktop than mobile, likely due to both faster average network speeds and faster processors. Only 38% of origins had good FCP on mobile. Render-blocking resources such as synchronous JavaScript can be a common culprit. Because TTFB is the first part of FCP, poor TTFB will make it difficult to achieve a good FCP.

**Note:** The thresholds for FCP have changed since last year. Be careful if you try to compare this year's data to last year's data.

422. <https://web.dev/fcp/>



*Figure 10.13. FCP performance by connection type*

Origins at 3G and below speeds experienced significant degradations in FCP. Again, ensure that you are profiling your website using real devices and networks that reflect your user data from analytics. Your JavaScript bundles may not seem significant when you're only profiling on high-end desktops with fiber connections.

Offline connections were closer in performance to 4G though not quite as good. Service worker start-up time plus multiple cache reads could have contributed. More factors come into play with FCP than with TTFB.



Figure 10.14. FCP performance by rank

Like TTFB, FCP improved with higher rankings. Also like TTFB, only 19.5% of origins on WordPress experienced good FCP performance<sup>423</sup>. Since their TTFB performance was poor, it is not surprising that their FCP is also slow. It's difficult to achieve good scores on FCP and LCP if TTFB is slow.

Common culprits for poor FCP are render-blocking resources, server response times (anything associated with a slow TTFB), large network payloads, and more.

## Largest Contentful Paint (LCP)

Largest Contentful Paint (LCP)<sup>424</sup> measures the time from start load to when the browser renders the largest image or text in the viewport.

423. <https://datastudio.google.com/s/kZ9KOd-sBQw>

424. <https://web.dev/lcp/>



Figure 10.15. LCP performance by device

LCP was faster on desktop than mobile. TTFB affects LCP like FCP. Comparisons by device, connection type, and rank all mirror the trends of FCP. Render-blocking resources, total weight, and loading strategies all affect LCP performance.



Figure 10.16. LCP performance by connection type

Offline origins with good LCP more closely matched 4G experiences, though poor LCP experiences were higher for offline. LCP occurs after FCP, and the additional budget of 0.7 seconds could be why more offline websites achieved good LCP than FCP.

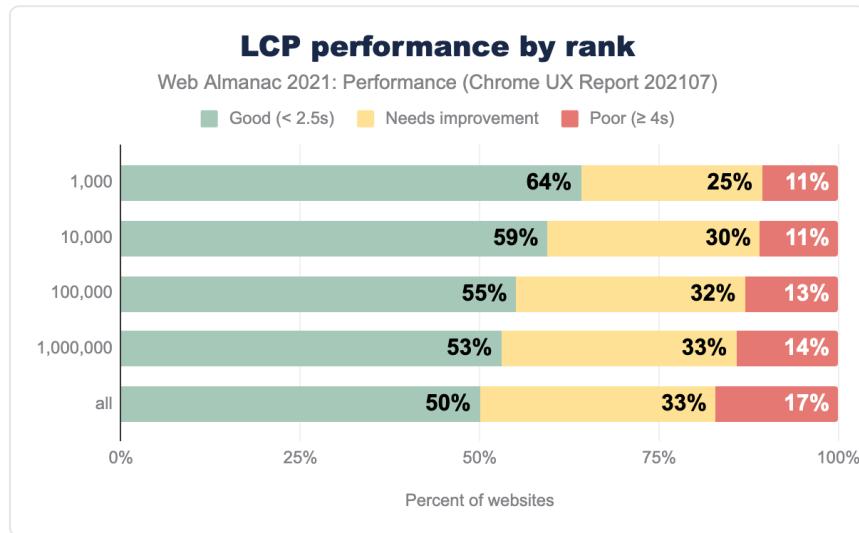


Figure 10.17. LCP performance by rank

For LCP, the differences in performance by rank were closer than FCP. Also, a higher proportion of origins in the top 1,000 had poor LCP. On WordPress, 28% of origins experienced good LCP<sup>425</sup>. This is an opportunity to improve user experience as poor LCP is usually caused by a handful of problems.

## The LCP Element

Let's take a deeper dive into the LCP element.

425. <https://datastudio.google.com/s/kvq1oJ60jaQ>

## Top 15 LCP HTML element nodes

Web Almanac 2021: Performance

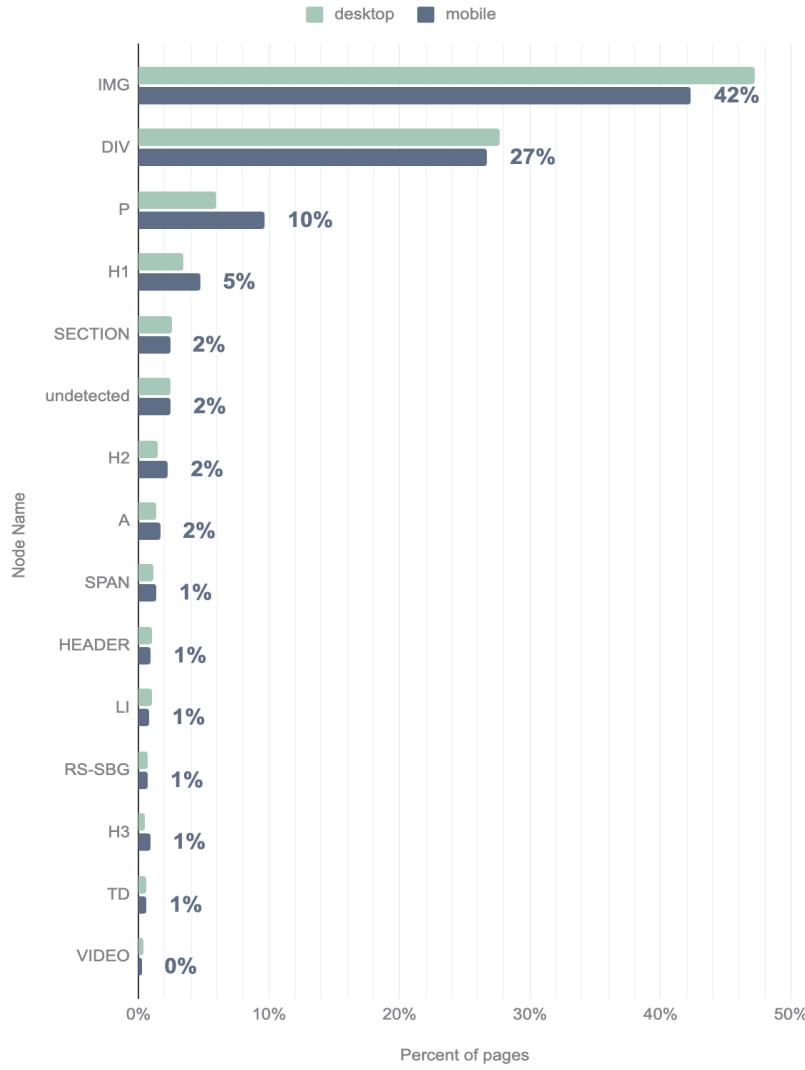


Figure 10.18. Top 15 LCP HTML element nodes

IMG, DIV, P, and H1 made up 83% of all LCP nodes (on mobile). This doesn't tell us if the content was an image or text, as background images can be applied with CSS.

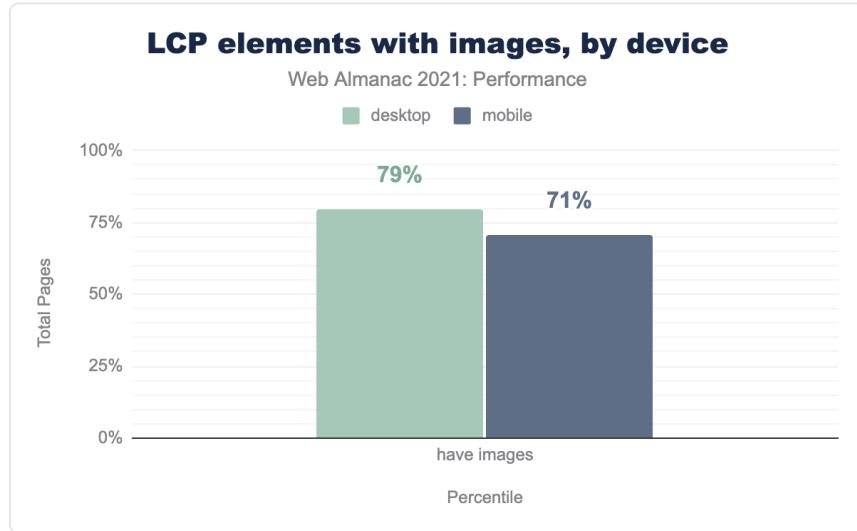


Figure 10.19. LCP elements with images, by device

We can see that 71-79% of pages had an LCP element that was an image, regardless of HTML node. Furthermore, desktop devices had a higher rate of LCPs as images. This could be due to less real estate on smaller screens pushing images out of the viewport resulting in heading text being the largest element.

In both cases, images comprised the majority of LCP elements. This warrants a deeper dive into how those images are loading.



Figure 10.20. LCP elements with potential performance anti-patterns

For user experience, we want LCP elements to load as fast as possible. User experience is why LCP was selected as one of the Core Web Vitals. We do not want it to be lazy-loaded as that further delays the render. However, we can see that 9.3% of pages used the native loading=lazy flag on the LCP `<img>` element.

Not all browsers support native lazy loading. Popular lazy loading polyfills detect a “lazyload” class on an image element. Thus, we can identify more possibly lazy-loaded images by adding images with a “lazyload” class to the total. The percent of sites probably lazy loading their LCP `<img>` element jumps up to 16.5% on mobile.

Lazy loading your LCP element will result in worse performance. *Don't do it!* WordPress was an early adopter of native lazy loading. The early method was a naive solution applying lazy loading to all images, and the results showed a negative performance correlation<sup>426</sup>. They were able to use this data to implement a more nuanced approach for better performance.

The `decode` attribute for images is relatively new. Setting it to `async` can improve load and scroll performance. Currently, 0.4% of sites used the `async` decode directive for their LCP image. The negative impact of asynchronous decode on an LCP image is currently unclear. Thus, test your site before and after if you choose to set an LCP image to `decode="async"`.

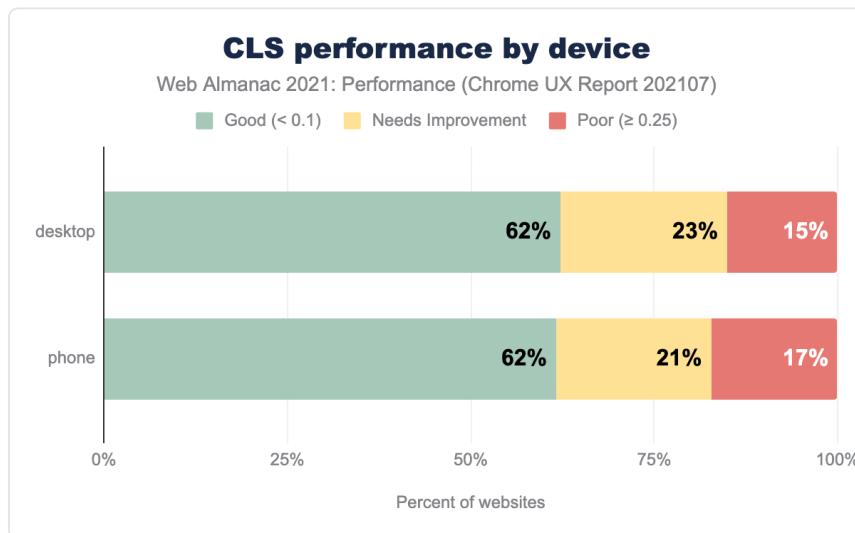
426. <https://web.dev/lcp-lazy-loading/>

# 354

*Figure 10.21. Websites attempted to use native lazy-loading on LCP elements that are not images or iframes*

Interestingly, 354 origins on desktop attempted to use native lazy-loading on HTML elements that do not support the loading attribute (e.g., `<div>`). The loading attribute is only supported on `<img>` and, in some browsers, `<iframe>` elements (see Can I use<sup>427</sup>).

## Cumulative Layout Shift (CLS)



*Figure 10.22. CLS performance by device*

Cumulative Layout Shift (CLS)<sup>428</sup> is characterized by how much layout shift a user experiences, not how long it takes to visually see something like FCP and LCP. As such, performance by device was fairly equivalent.

427. <https://caniuse.com/loading-lazy-attr>

428. <https://web.dev/cls/>



Figure 10.23. CLS performance by connection type

Performance degradation from 4G to 3G and below was not as pronounced as with FCP and LCP. Some degradation exists, but it's not reflected in the device data, only the connection type.

Offline websites had the highest CLS performance of all connection types. For sites with service worker caching, some assets like images and ads that would otherwise cause layout shifts may not be cached. Thus, they would never load and never cause a layout shift. Often fallback HTML for these sites can be more basic versions of the online website.



Figure 10.24. CLS performance by rank

For ranking, CLS performance showed an interesting trough for the top 10,000 websites. In addition, all the ranked groups above 1M performed worse than the sites ranked under 1M. Since the “all” group had better performance than all the other ranked groupings the sub-1M group performs better. WordPress may again play a role in this as 60% of origins on WordPress experienced a good CLS<sup>429</sup>.

Common culprits for poor CLS include not reserving space for images, text shifts when web fonts are loaded, top banners inserted after first paint, non-composited animations, and iframes.

## First Input Delay (FID)

First Input Delay (FID)<sup>430</sup> measures the time from when a user first interacts with a page to the time the browser begins processing event handlers in response to that interaction.

429. <https://datastudio.google.com/s/qG00yMxSa3o>

430. <https://web.dev/fid/>

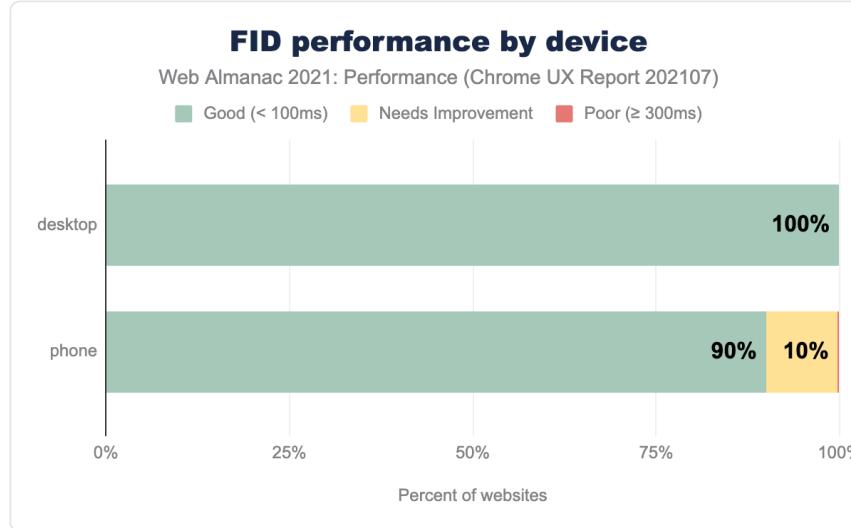


Figure 10.25. FID performance by device

FID performance was better on desktop than on mobile devices likely due to device speeds which can better handle larger amounts of JavaScript.



Figure 10.26. FID performance by connection type

FID performance degraded some by connection type, but less so than the other metrics. The

high distribution of scores seemed to reduce the amount of variance in the results.

Unlike the other metrics, FID was worse for offline websites than any other connection category. This could be due to the more complex nature of many websites with service workers. Having a service worker does not eliminate the impact of client-side JavaScript running on the main thread.



Figure 10.27. FID performance by rank

FID performance by rank was flat.

For all FID metrics, we see very large bars in the “good” category which makes it less effective unless we’ve truly hit peak performance. The good news is the Chrome team is evaluating this now<sup>431</sup> and would like your feedback.

If your site’s performance is not in the “good” category, then you definitely have a performance problem. A common culprit for FID issues is too much long-running JavaScript. Keep your bundle sizes small and pay attention to third-party scripts.

<sup>431</sup> <https://web.dev/better-responsiveness-metric/>

## Total Blocking Time (TBT)

*The Total Blocking Time (TBT) metric measures the total amount of time between First Contentful Paint (FCP) and Time to Interactive (TTI) where the main thread was blocked for long enough to prevent input responsiveness.*

– Web.dev<sup>432</sup>

Total Blocking Time (TBT)<sup>433</sup> is a lab-based metric that helps us debug potential interactivity issues. FID is a field-based metric, and TBT is its lab-based analog. Currently, when evaluating client websites, I reach for total blocking time TBT as another indicator of possible performance issues due to JavaScript.

Unfortunately, TBT is not measured in the Chrome User Experience Report. But, we can still get an idea of what's going on using the HTTP Archive Lighthouse data (only collected for mobile):

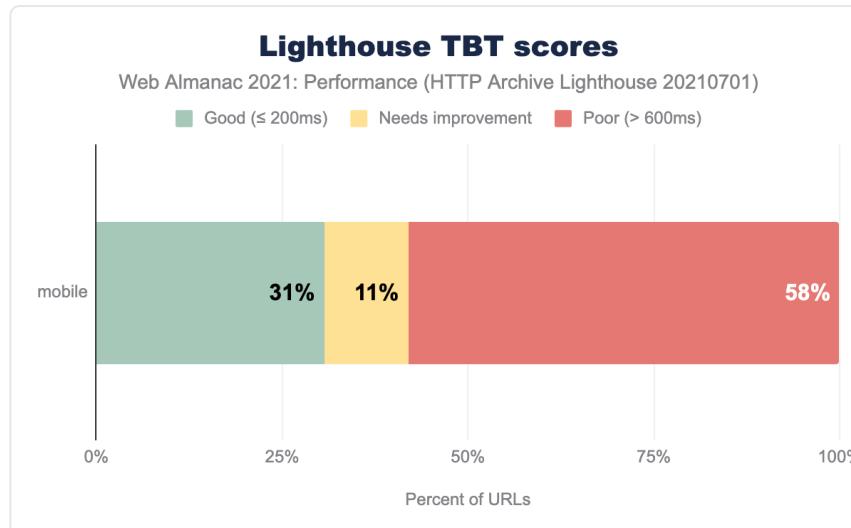


Figure 10.28. Lighthouse TBT scores

**Note:** The groups in the chart are based off of the Lighthouse score for TBT (e.g.,  $\geq 0.9$  results in “good”). Due to rounding of the score, some TBT values slightly above 200ms get categorized as “good” (and similarly at the 600ms threshold).

432. <https://web.dev/tbt/>

433. <https://web.dev/tbt/>

Remember that the data is a single, throttled-CPU Lighthouse run through WebPageTest and does not reflect real user experiences. Yet, potential interactivity looked much worse when looking at TBT versus FID. The “real” evaluation of your interactivity is probably somewhere between. Thus, if your FID is “good”, take a look at TBT in case you’re missing some poor user experiences that FID can’t catch yet. The same issues that cause poor FID also cause poor TBT.

## 67 seconds

*Figure 10.29. Longest TBT*

## Conclusion

Performance improved since 2020. Though we still have a long way to go to provide great user experience, we can take steps to improve it.

First, you cannot improve performance unless you can measure it. A good first step here is to measure your site using real user devices and to set up real-user monitoring (RUM). You can get a flavor of how your site performs with Chrome users with the CrUX dashboard launcher<sup>434</sup> (if your site is in the dataset). You should set up a RUM solution that measures across multiple browsers. You can build this yourself or use one of many analytics vendors’ solutions.

Second, as new features in HTML, CSS, and JavaScript are released, make sure you understand them before implementing them. Use A/B testing to verify that adopting a new strategy results in improved performance. For example, don’t lazy-load images above the fold. If you have a RUM tool implemented, you can better detect when your changes accidentally cause regressions.

Third, continue to optimize for both FID (field/real-user data) and TBT (lab data). Take a look at the proposal<sup>435</sup> for a new responsiveness metric and participate by providing feedback. A new animation smoothness metric<sup>436</sup> is also being proposed. In our quest for a faster web, change is inevitable and for the better. As we continue to optimize, you’re participation is key.

Finally, we saw that WordPress can impact the performance of the top 10M websites, and maybe more. This is a lesson that every CMS and framework should heed. The more we can set up smart defaults for performance at the framework level, the better we can make the web while also make developers’ jobs easier.

What did you find most interesting or surprising? Share your thoughts with us on Twitter

434. <https://rvisconti.github.io/crux-dash-launcher/>

435. <https://web.dev/responsiveness/>

436. <https://web.dev/smoothness/>

(@HTTPArchive)!

---

## Author

---



### Sia Karamalegos

 @TheGreenGreek  siakaramalegos  karamalegos  <https://sia.codes>

Sia Karamalegos is a web developer, international conference speaker, and writer. She is a Google Developer Expert in Web Technologies, a Cloudinary Media Developer Expert, a Stripe Community Expert, and co-organizes the Eleventy Meetup. Check out her writing, speaking, and newsletter on [sia.codes<sup>437</sup>](https://sia.codes) or find her on Twitter<sup>438</sup>.

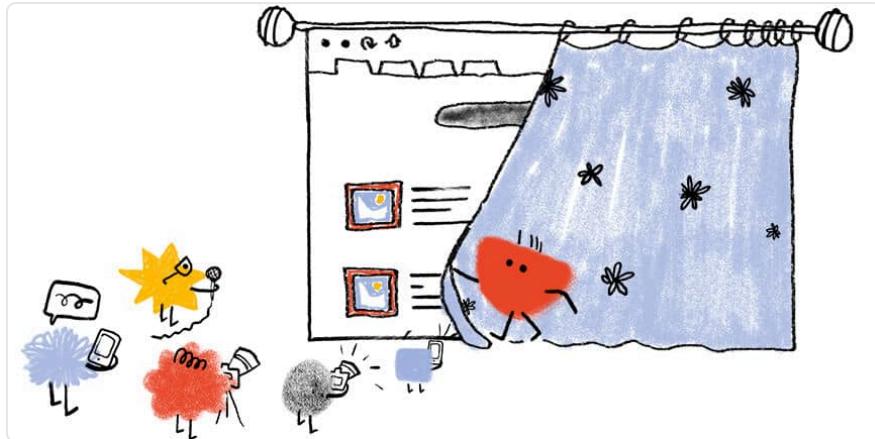
---

---

437. <https://sia.codes/>  
438. <https://twitter.com/thegreengreek>

## Part II Chapter 11

# Privacy



**Written by Yana Dimova and Victor Le Pochat**

**Reviewed by Maud Nalpas**

**Analyzed by Victor Le Pochat and Max Ostapenko**

**Edited by Barry Pollard**

## Introduction

*“On the Internet, nobody knows you’re a dog.”* While it might be true that you could try to remain anonymous to use the Internet as such, it can be quite hard to keep your personal data fully private.

A whole industry<sup>439</sup> is dedicated to tracking users online, to build detailed user profiles for purposes such as targeted advertising, fraud detection, price differentiation, or even credit scoring. Sharing geolocation data with websites can prove very useful in day-to-day life, but may also allow companies to see your every movement<sup>440</sup>. Even if a service treats a user’s private information diligently, the mere act of storing personal data provides hackers with an opportunity to breach services and leak millions of personal records online<sup>441</sup>.

439. <https://crackedlabs.org/en/corporate-surveillance/>

440. <https://www.nytimes.com/interactive/2019/12/19/opinion/location-tracking-cell-phone.html>

441. <https://haveibeenpwned.com/>

Recent legislative efforts such as the GDPR<sup>442</sup> in Europe, CCPA<sup>443</sup> in California, LGPD<sup>444</sup> in Brazil, or the PDP Bill<sup>445</sup> in India all strive to require companies to protect personal data and implement privacy by default, including online. Major technology companies such as Google, Facebook and Amazon have already received massive fines<sup>446</sup> for alleged violations of user privacy.

These new laws have given users a much larger say in how comfortable they are with sharing personal data. You probably already have clicked through quite a few cookie consent banners that enable this choice. Furthermore, web browsers are implementing technological solutions<sup>447</sup> to improve user privacy, from blocking third-party cookies over hiding sensitive data to innovative ways to balance legitimate use cases on personal attributes with individual user privacy.

In this chapter, we give an overview of the current state of privacy on the web. We first consider how user privacy can be harmed: we discuss how websites profile you through online tracking, and how they access your sensitive data. Next, we dive into ways websites protect sensitive data and give you a choice through privacy preference signals. We close with an outlook on the efforts that browsers are making to safeguard your privacy in the future.

## How websites profile you: online tracking

The HTTP protocol is inherently stateless, so by default there is no way for a website to know whether two visits to two different websites, or even two visits to the same website, are from the same user. However, such information could be useful for websites to build more personalized user experiences, and for third parties building profiles of user behavior across websites to fund content on the web through targeted advertising or providing services such as fraud detection.

Unfortunately, obtaining this information currently often relies on online tracking, around which many large and small companies have built their business<sup>448</sup>. This has even led to calls to ban targeted advertising<sup>449</sup>, since invasive tracking is at odds with users' privacy. Users might not want anyone to follow their tracks across the web—especially when visiting websites on sensitive topics. We'll look at the main companies and technologies that make up the online tracking ecosystem.

---

442. <https://ec.europa.eu/info/law/law-topic/data-protection/data-protection-eu>

443. <https://www.oag.ca.gov/privacy/ccpa>

444. <https://www.gov.br/cidadania/pt-br/acesso-a-information/lgpd>

445. <https://www.mca.gov.in/data-protection-framework>

446. [https://en.wikipedia.org/wiki/GDPR\\_fines\\_and\\_notices](https://en.wikipedia.org/wiki/GDPR_fines_and_notices)

447. <https://privacysandbox.com/>

448. <https://crackedlabs.org/en/corporate-surveillance/>

449. <https://www.forbrukerradet.no/wp-content/uploads/2021/06/20210622-final-report-time-to-ban-surveillance-based-advertising.pdf>

## Third-party tracking

Online tracking is often done through third-party libraries. These libraries usually provide some (useful) service, but in the process some of them also generate a unique identifier for each user, which can then be used to follow and profile users across websites. The WhoTracksMe<sup>450</sup> project is dedicated to discovering the most widely deployed online trackers. We use WhoTracksMe's classification of trackers but restrict ourselves to four categories<sup>451</sup>, because they are the most likely to cover services where tracking is part of the primary purpose: *advertising, pornvertising, site analytics and social media*.

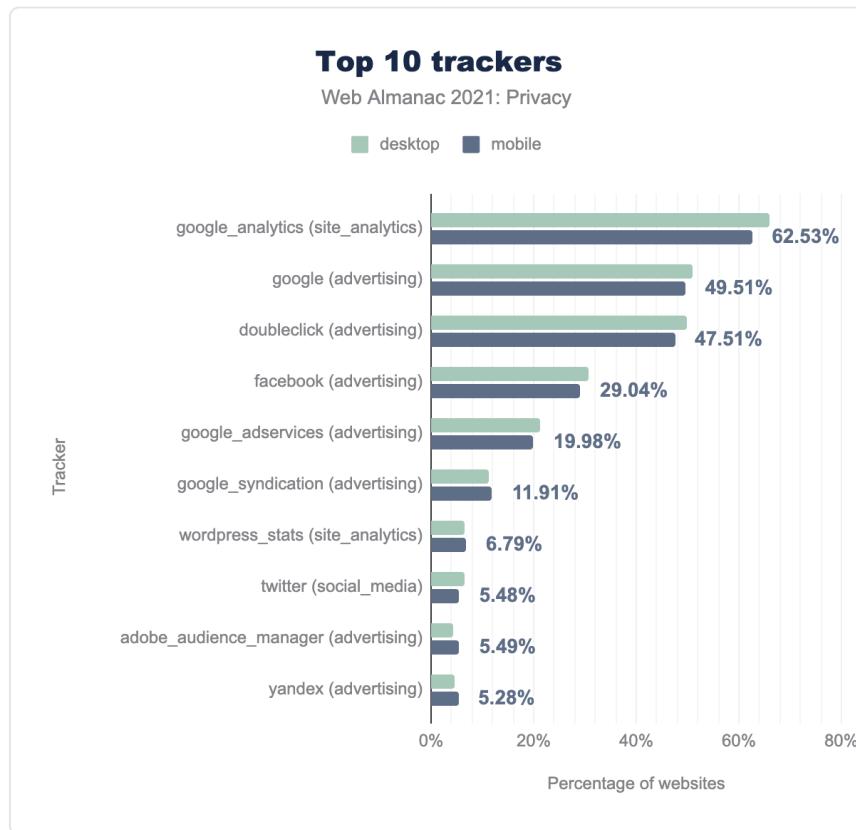


Figure 11.1. 10 most popular trackers and their prevalence.

We see that Google-owned domains are prevalent in the online tracking market. Google Analytics, which reports website traffic, is present on almost two-thirds of all websites. Around

450. <https://whotracks.me/>

451. [https://whotracks.me/blog/tracker\\_categories.html](https://whotracks.me/blog/tracker_categories.html)

30% of sites include Facebook libraries, while other trackers only reach single-digit percentages.



Figure 11.2. Most common tracker categories.

Overall, 82.08% of mobile sites and 83.33% of desktop sites include at least one tracker, usually for site analytics or advertising purposes.



Figure 11.3. The number of trackers per website.

Three out of four websites have fewer than 10 trackers, but there is a long tail of sites with many more trackers: one desktop site contacted 133 (!) distinct trackers.

### Third-party cookies

The main technical approach to store and retrieve cross-site user identifiers is through cookies that are persistently stored in your browser. Note that while third-party cookies are often used for cross-site tracking, they can also be used for non-tracking use cases, like state sharing for a third-party widget across sites. We searched for the cookies that appear most often while browsing the web, and the domains that set them.



Figure 11.4. Top 10 domains setting cookies from headers.

Google's subsidiary DoubleClick takes the top spot by setting cookies on 31.4% of desktop websites and 28.7% on mobile websites. Another major player is Facebook, which stores cookies on 21.4% of mobile websites. Most of the other top domains setting cookies are related to online advertising.



Figure 11.5. Top 10 cookies set from headers.

Looking at the specific cookies that these websites set, the most common cookie from a tracker is the `test_cookie` from doubleclick.net. The next most common cookies are advertising-related and remain on a user's device much longer: Facebook's `fr` cookie persists for 90 days<sup>452</sup>, while DoubleClick's `IDE` cookie stays for 13 months in Europe and 2 years elsewhere<sup>453</sup>.

With `Lax` becoming the default value of the `SameSite` cookie attribute, sites that want to continue sharing third-party cookies across websites must explicitly set this attribute to `None`. For third parties, 85% have done this so far on mobile and 64% on desktop, potentially for tracking purposes. You can read more about the `SameSite` cookie attribute over at the Security chapter.

<sup>452</sup> <https://www.facebook.com/policy/cookies/>

<sup>453</sup> <https://business.safety.google/adscookies/>

## Fingerprinting

With the rise of privacy-protecting tools such as ad blockers and initiatives to phase out third-party cookies from major browsers such as Firefox<sup>454</sup>, Safari<sup>455</sup>, and by 2023 also Chrome<sup>456</sup>, trackers are looking for more persistent and stealthy ways to track users across sites.

One such technique is *browser fingerprinting*. A website collects information about the user's device, such as the user agent<sup>457</sup>, screen resolution and installed fonts, and uses the often unique combination of those values to create a *fingerprint*. This fingerprint is recreated every time a user visits the website and can then be matched to identify the user. While this method can be used for fraud detection, it is also used to persistently track recurring users, or to track users across sites.

Detecting fingerprinting is complex: it is effective through a combination of method calls and event listeners that may also be used for non-tracking purposes. Instead of focusing on these individual methods, we therefore focus on five popular libraries that make it easy for a website to implement fingerprinting.



Figure 11.6. Websites using each fingerprinting library.

From the percentage of websites using these third-party services, we can see that the most

454. <https://blog.mozilla.org/en/products/firefox/todays-firefox-blocks-third-party-tracking-cookies-and-cryptomining-by-default/>

455. <https://webkit.org/blog/10218/full-third-party-cookie-blocking-and-more/>

456. <https://blog.google/products/chrome/updated-timeline-privacy-sandbox-milestones/#:-:text=Chrome%20could%20then%20phase%20out%20third-party%20cookies%20over%20a%20three%20month%20period%2C%20starting%20in%20mid-2023%20and%20ending%20in%20late%202023>

457. [https://developer.mozilla.org/docs/Glossary/User\\_agent](https://developer.mozilla.org/docs/Glossary/User_agent)

widely used library, Fingerprint.js<sup>458</sup>, is used 19 times more on desktop than the second most popular library. However, the overall percentage of websites that use an external library to fingerprint their users is quite small.

## CNAME tracking

Continuing with techniques that circumvent blocks on third-party tracking, CNAME tracking<sup>459</sup> is a novel approach where a first-party subdomain masks the use of a third-party service using a CNAME record at the DNS level<sup>460</sup>. From the viewpoint of the browser, everything happens within a first-party context, so none of the third-party countermeasures are applied. Major tracking companies such as Adobe and Oracle are already offering CNAME tracking solutions to their customers. For the results on CNAME-based tracking included in this chapter, we refer to research<sup>461</sup> completed by one of this chapter's authors (and others) where they developed a method to detect CNAME-based tracking, based on DNS data and request data from HTTP Archive.

---

458. <https://fingerprintjs.com/>  
459. <https://medium.com/nextdns/cname-cloaking-the-dangerous-disguise-of-third-party-trackers-195205dc522a>  
460. <https://adguard.com/en/blog/cname-tracking.html>  
461. <https://sciendo.com/article/10.2478/popets-2021-0053>

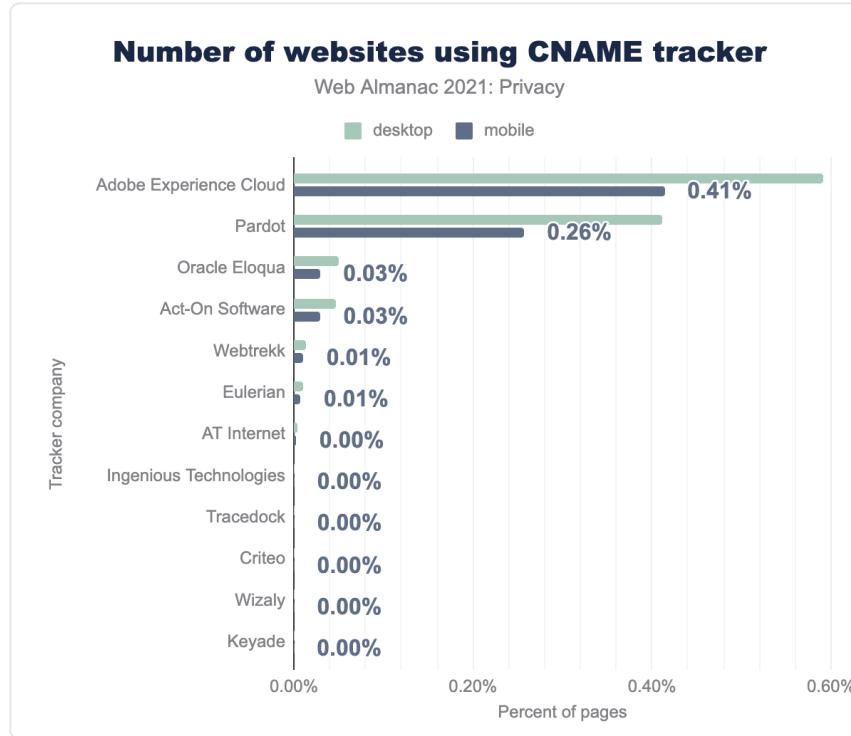


Figure 11.7. Websites using CNAME-based tracking on a desktop client.

The most popular company performing CNAME-based tracking is Adobe, which is present on 0.59% of desktop websites, and 0.41% of mobile websites. Also notable in size is Pardot<sup>462</sup>, with 0.41% and 0.26% respectively.

Those numbers may seem a small percentage, but that opinion changes when segregating the data by site popularity.

<sup>462</sup> <https://www.pardot.com/>

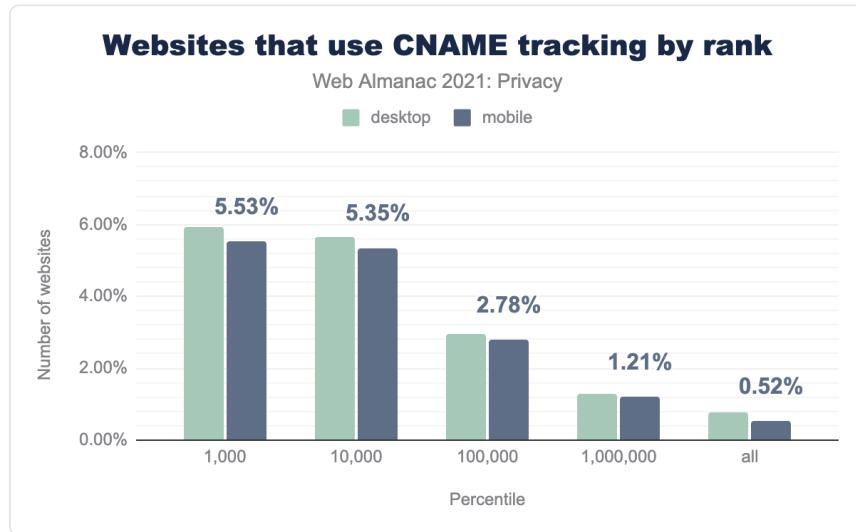


Figure 11.8. Websites that use CNAME tracking by rank.

When we look at the rank of the websites that use CNAME-based tracking, we see that 5.53% of the top 1,000 websites on mobile embed a CNAME tracker. In the top 100,000, that number falls to 2.78% of websites, and when looking at the full data set it falls to 0.52%.

## Public suffix of sites with CNAME-based tracking

Web Almanac 2021: Privacy

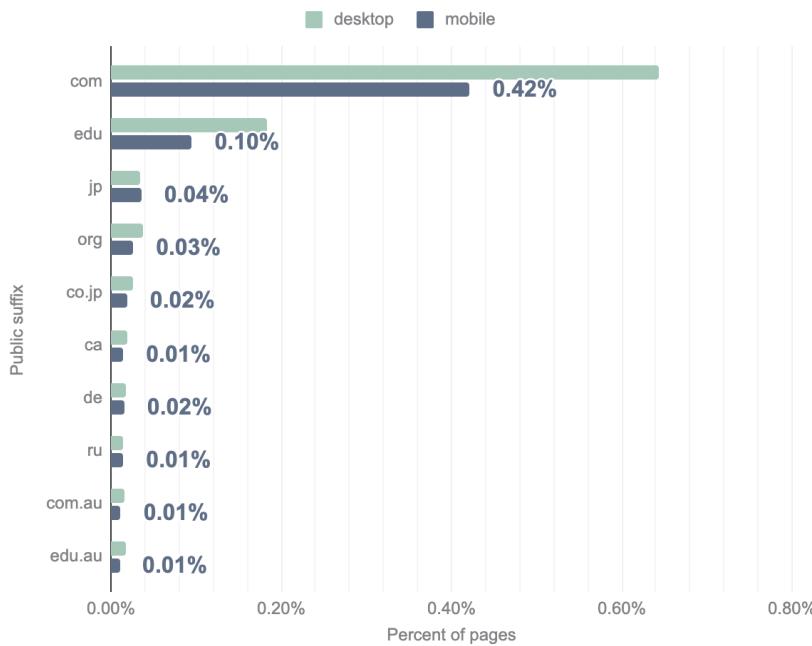


Figure 11.9. Public suffix of sites with CNAME-based tracking.

Apart from the `.com` suffix, a large number of the websites using CNAME-based tracking have a `.edu` domain. Also, a notable amount of CNAME trackers are prevalent on `.jp` and `.org` websites.

CNAME-based tracking can be a countermeasure to when the user might have enabled tracking protection against third-party tracking. Since few tracker-blocking tools and browsers<sup>463</sup> have already implemented a defense against CNAME tracking, it is prevalent on a number of websites up to date.

## (Re)targeting

Advertisement retargeting refers to the practice of keeping track of the products that a user has looked at but has not purchased and following up with ads about these products on

<sup>463</sup> <https://www.cookiestatus.com/>

different websites. Instead of opting for an aggressive marketing strategy while the user is visiting, the website chooses to nudge the user into buying the product by continuously reminding them of the brand and product.

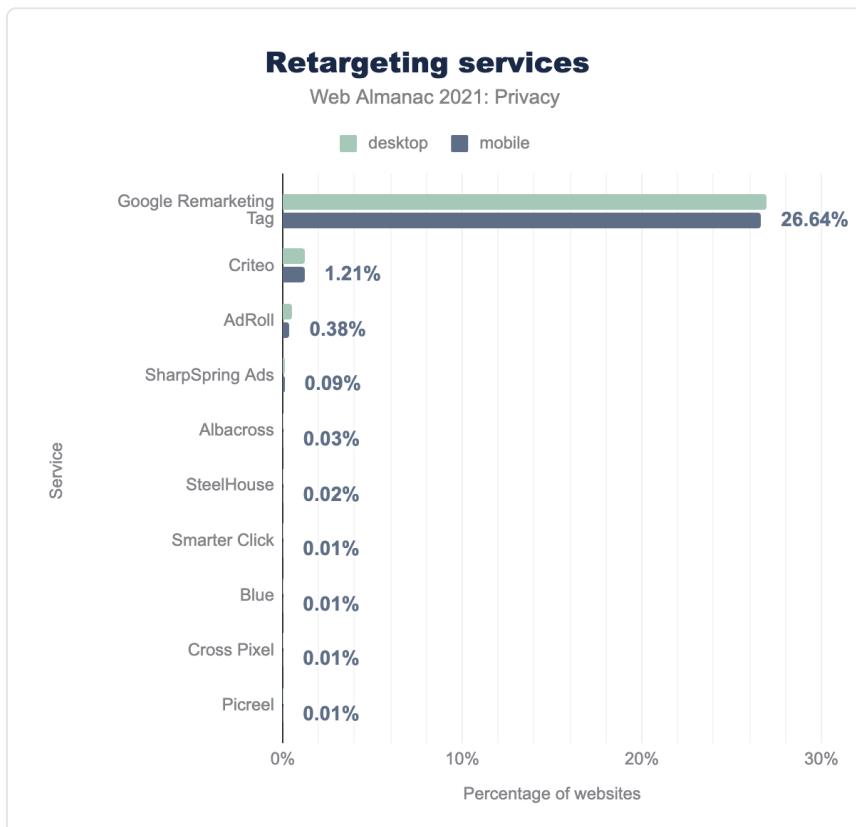


Figure 11.10. Percentage of pages using a retargeting service.

A number of trackers provide a solution for ad retargeting. The most widely used one, Google Remarketing Tag, is present on 26.92% of websites on desktop and 26.64% of websites on mobile, far and above all other services which are used by less than 1.25% of sites each.

## How websites handle your sensitive data

Some websites request access to specific features and browser APIs that can impact the user's privacy, for instance by accessing the geolocation data, microphone, camera, etc. These features usually serve very useful purposes, such as discovering nearby points of interest or

allowing people to communicate with each other. While these features are only activated when a user consents, there is a risk of exposing sensitive data if the user does not fully understand how those resources are used, or if a site misbehaves.

We looked at how often websites request access to sensitive resources. Moreover, any time a service stores sensitive data, there is the danger of hackers stealing and leaking that data. We'll look at recent data breaches that prove that this danger is real.

## Device sensors

Sensors can be useful to make a website more interactive but could also be abused for fingerprinting users<sup>464</sup>. Based on the use of JavaScript event listeners, the orientation of the device is accessed the most, both on mobile and on desktop clients. Note that we searched for the presence of event listeners on websites, but we do not know if the code is actually executed. Therefore, the access to device sensor events in this section is an upper bound.

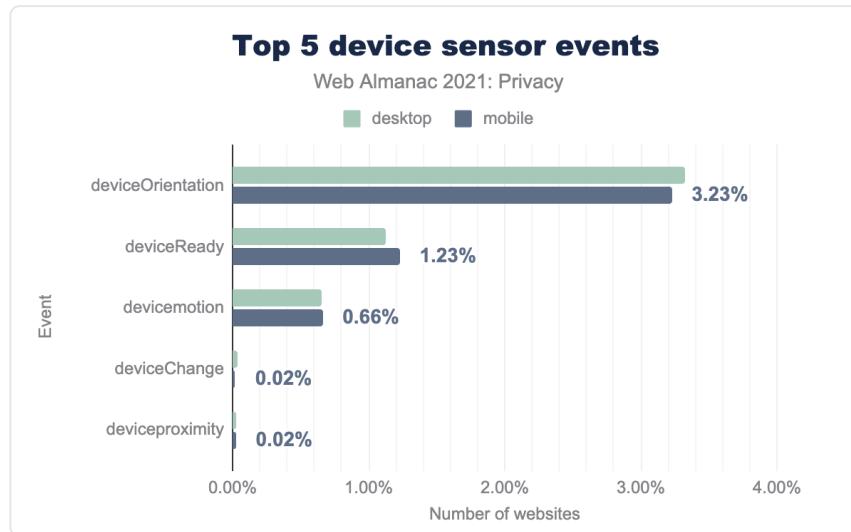


Figure 11.11. 5 most used sensor events.

## Media devices

The MediaDevices API<sup>465</sup> can be used to access connected media input such as cameras, microphones and screen sharing.

464. <https://www.esat.kuleuven.be/cosic/publications/article-3078.pdf>

465. <https://developer.mozilla.org/docs/Web/API/MediaDevices>

# 7.23%

Figure 11.12. Percent of desktop pages that used the `MediaDevices EnumerateDevices()` API.

On 7.23% of desktop websites, and 5.33% of mobile websites the `enumerateDevices()` method is called, which provides a list of the connected input devices.

## Geolocation-as-a-service

Geolocation services provide GPS and other location data (such as IP address<sup>466</sup>) of the user and can be used by trackers to provide more relevant content to the user among other things.

Therefore, we analyze the use of “geolocation-as-a-service” technologies on websites, based on libraries detected through Wappalyzer.

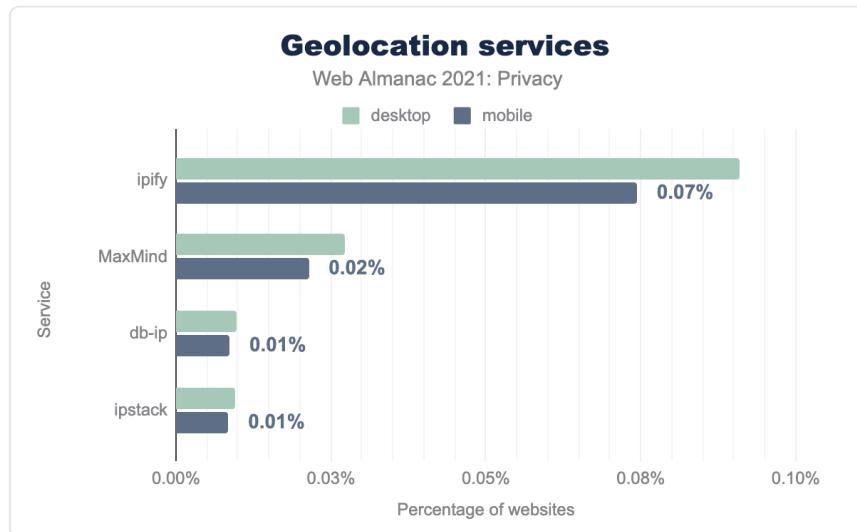


Figure 11.13. Percentage of websites that use geolocation services.

We find that the most popular service, ipify<sup>467</sup>, is used on 0.09% of desktop websites and 0.07% of mobile websites. So, it would appear that few websites use geolocation services.

466. [https://developer.mozilla.org/docs/Glossary/IP\\_Address](https://developer.mozilla.org/docs/Glossary/IP_Address)

467. <https://www.ipify.org/>

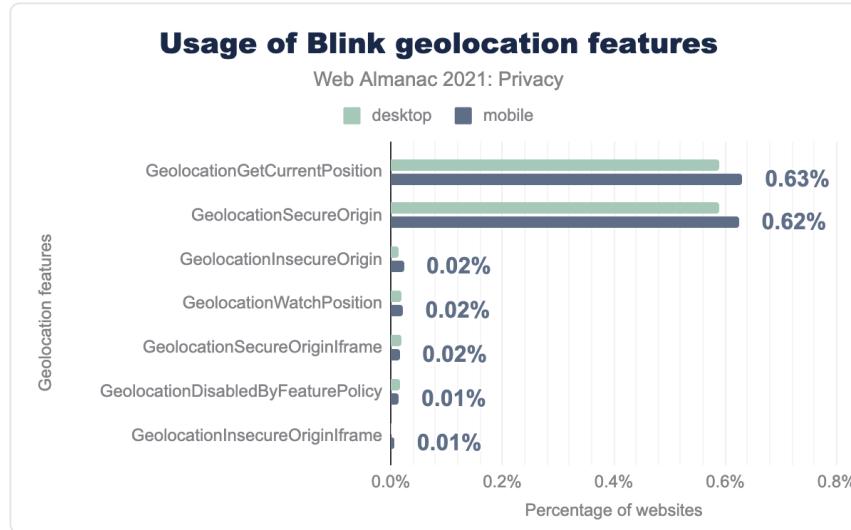


Figure 11.14. Percentage of websites that use geolocation features.

Geolocation data can also be accessed by websites through a web browser API<sup>468</sup>. We find that 0.59% of websites on a desktop client and 0.63% of websites on a mobile client access the current position of the user (based on Blink features).

## Data breaches

Poor security management within a company can have a significant impact on its customers' private data. Have I Been Pwned<sup>469</sup> allows users to check whether their email address or phone number was leaked in a data breach. At the time of this writing, Have I Been Pwned has tracked 562 breaches, leaking 640 million records. In 2020 alone, 40 services were breached and personal data about millions of users leaked. Three of these breaches were marked as *sensitive*, referring to the possibility of a negative impact on the user if someone were to find that user's data in the breach. One example of a sensitive breach is "Carding Mafia"<sup>470</sup>, a platform where stolen credit cards are traded.

*Note that 40 breaches in the previous year is a lower bound, since many breaches are only discovered, or made public, several months after they have occurred.*

468. [https://developer.mozilla.org/docs/Web/API/Geolocation\\_API](https://developer.mozilla.org/docs/Web/API/Geolocation_API)

469. <https://haveibeenpwned.com/>

470. <https://www Vice.com/en/article/v7m9jx/credit-card-hacking-forum-gets-hacked-exposing-300000-hackers-accounts>



Figure 11.15. Number of impacted accounts in breaches per data class. (Source: *Have I Been Pwned*<sup>471</sup>)

Every data breach tracked by *Have I Been Pwned*<sup>472</sup> leaks email addresses, since this is how users query whether their data was breached. Leaked email addresses are already a huge privacy risk, since many users employ their full name or credentials to set up their email address. Furthermore, a lot of other highly sensitive information is leaked in some breaches, such as users' genders, bank account numbers and even full physical addresses.

## How websites protect your sensitive data

While you're browsing the web, there is certain data that you might want to keep private: the web pages that you visit, any sensitive data that you enter into forms, your location, and so on. Over at the Security chapter, you can learn how 91.1% of mobile sites have enabled HTTPS to protect your data from snooping while it traverses the Internet. Here, we'll focus on how websites can further instruct browsers to ensure privacy for sensitive resources.

### Permissions Policy / Feature Policy

The Permissions Policy<sup>473</sup> (previously called Feature Policy) provides a way for websites to

471. <https://haveibeenpwned.com/>

472. <https://haveibeenpwned.com/>

473. <https://www.w3.org/TR/permissions-policy-1/>

define which web features they intend to use, and which features will need to be explicitly approved by the user—when requested by third parties for instance. This gives websites control over what features embedded third-party scripts can request to access. For example, a permissions policy can be used by a website to ensure that no third-party requests microphone access on their site. The policy allows developers to granularly choose web APIs they intend to use, by specifying them with the `allow` attribute.

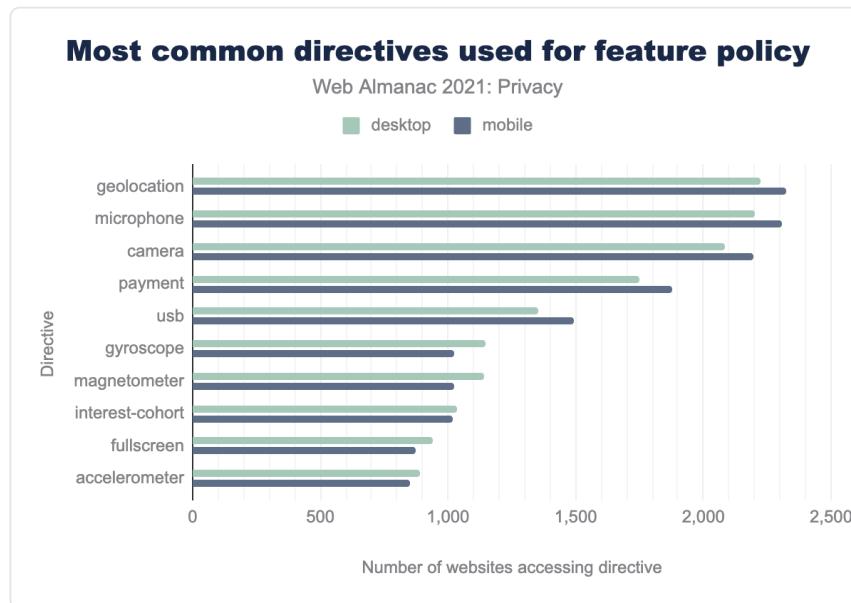


Figure 11.16. Number of websites accessing a feature policy directive.

The most commonly used directives with relation to the feature policy are shown above. On 3,049 websites on mobile and 2,901 websites on desktop, the use of the microphone feature is specified. A tiny subset of our dataset, showing this is still a niche technology. Other often restricted features are geolocation, camera and payment.

To gain a deeper understanding of how the directives are used, we looked at the top 3 most used directives and the distribution of the values assigned to these directives.



Figure 11.17. Values used for the 3 most popular feature policy directives.

`none` is the most used value. This specifies that the feature is disabled in top-level and nested browsing contexts. The second most used value, `self` is used to specify that the feature is allowed in the current document and within the same origin, while `*` allows full, cross-origin access.

## Referrer Policy

HTTP requests may include the optional `Referer` header, which indicates the origin or web page URL a request was made from. The `Referer` header might be present in different types of requests:

- Navigation requests, when a user clicks a link.
- Subresource requests, when a browser requests images, iframes, scripts, and other resources that a page needs.

For navigations and iframes, this data can also be accessed via JavaScript using `document.referrer`.

The `Referer` value can be insightful. But when the full URL including the path and query string is sent in the `Referer` across origins, this can be privacy-hindering: URLs can contain private information—sometimes even identifying or sensitive information. Leaking this silently across origins can compromise users' privacy and pose security risks. The `Referrer-Policy`

HTTP header allows developers to restrict what referrer data is made available for requests made from their site to reduce this risk.



Figure 11.18. Percentage of websites that specify a Referrer Policy.

A first point to note is that most sites do not explicitly set a Referrer Policy. Only 11.12% of desktop websites and 10.38% of mobile websites explicitly define a Referrer Policy. The rest of them (the other 88.88% on desktop and 89.62% on mobile) will fall back to the browser's default policy. Most major browsers<sup>474</sup> recently introduced a default policy of `strict-origin-when-cross-origin`, such as Chrome<sup>475</sup> in August 2020 and Firefox<sup>476</sup> in March 2021. `strict-origin-when-cross-origin` removes the path and query fragments of the URL on cross-origin requests, which reduces security and privacy risks.

474. <https://web.dev/referrer-best-practices/#default-referrer-policies-in-browsers>

475. <https://developers.google.com/web/updates/2020/07/referrer-policy-new-chrome-default>

476. <https://blog.mozilla.org/security/2021/03/22/firefox-87-trims-http-referrers-by-default-to-protect-user-privacy/>

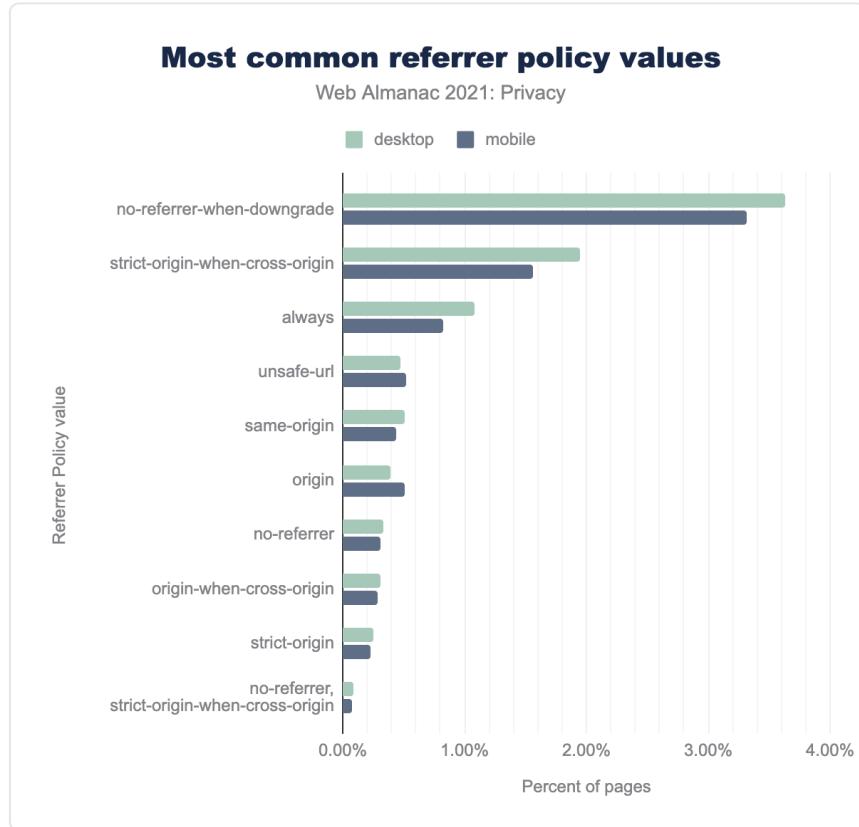


Figure 11.19. Percentage of pages using Referrer Policy values.

The most common Referrer Policy that is explicitly set is `no-referrer-when-downgrade`. It's set on 3.38% of websites on mobile clients and 3.81% of websites on desktop clients. `no-referrer-when-downgrade` is not privacy-enhancing. With this policy, full URLs of pages a user visits on a given site are shared in cross-origin HTTPS requests (the vast majority of requests), which makes this information accessible to other parties (origins).

In addition, around 0.5% of websites set the value of the referrer policy to `unsafe-url`, which allows the origin, host and query string to be sent with *any* request, regardless of the security level of the receiver. In this case, a referrer could be sent in the clear, potentially leaking private information. Worryingly, sites are actively being configured to enable this behavior.

*Note: Websites may also send the referrer information as a URL parameter to the destination site. We did not measure usage of that mechanism for this report.*

## User-Agent Client Hints

When a web browser makes an HTTP request, it will include a `User-Agent` header that provides information about the client's browser, device and network capabilities. However, this can be abused for profiling users or uniquely identifying them through fingerprinting.

User-Agent Client Hints<sup>477</sup> enable access to the same information as the `User-Agent` string, but in a more privacy-preserving way. This will in turn enable browsers to eventually reduce the amount of information provided by default by the `User-Agent` string, as Chrome is proposing with a gradual plan for User Agent Reduction<sup>478</sup>.

Servers can indicate their support for these Client Hints by specifying the `Accept-CH` header. This header lists the attributes that the server requests from the client in order to serve a device-specific or network-specific resource. In general, Client Hints provide a way for servers to obtain only the minimum information necessary to serve content in an efficient manner.



Figure 11.20. Percentage of pages that use User-Agent Client Hints.

However, at this point, few websites have implemented Client Hints. We also see a big difference between the use of Client Hints on popular websites and on less popular ones. 3.67% of the top 1,000 most popular websites on mobile request Client Hints. In the top 10,000 websites, the implementation rate drops to 1.44%.

477. <https://wicg.github.io/ua-client-hints/>

478. <https://www.chromium.org/updates/ua-reduction>

## How websites give you a privacy choice: Privacy preference signals

In light of the recent introduction of privacy regulations, such as those mentioned in the introduction, websites are required to obtain explicit user consent about the collection of personal data for any non-essential features such as marketing and analytics.

Therefore, websites turned to the use of cookie consent banners, privacy policies and other mechanisms (which have evolved over time<sup>479</sup>) to inform users about what data these sites process, and give them a choice. In this section, we look at the prevalence of such tools.

### Consent Management Platforms

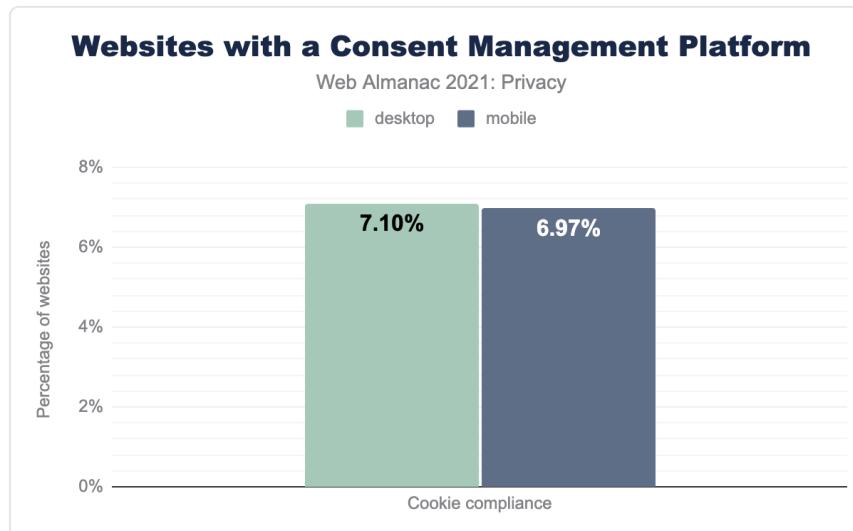


Figure 11.21. Percentage of websites that use a Consent Management Platform.

Consent Management Platforms (CMPs) are third-party libraries that websites can include to provide a cookie consent banner for users. We saw around 7% of websites using a Consent Management Platform.

479. <https://sciendo.com/article/10.2478/popets-2021-0069>



Figure 11.22. 10 most popular consent management platforms.

The most popular libraries are CookieYes<sup>480</sup> and Osano<sup>481</sup>, but we found more than twenty different libraries that allow websites to include cookie consent banners. Each library was only present on a small share of websites, at less than 2% each.

## IAB's Consent Frameworks

The Transparency and Consent Framework<sup>482</sup> (TCF) is an initiative of the Interactive Advertising Bureau Europe (IAB) for providing an industry standard for communicating user consent to advertisers. The framework consists of a Global Vendor List<sup>483</sup>, in which vendors can specify the legitimate purpose of the processed data, and a list of CMPs who act as an intermediary between the vendors and the publishers. Each CMP is responsible for communicating the legal basis and storing the consent option provided by the user in the browser. We refer to the stored cookie as the *consent string*.

TCF is meant as a GDPR-compliant mechanism in Europe, although a recent decision by the Belgian Data Protection Authority<sup>484</sup> found that this system is still infringing. When the CCPA

480. <https://www.cookieyes.com/>

481. <https://www.osano.com/>

482. <https://iabeurope.eu/transparency-consent-framework/>

483. <https://iabeurope.eu/vendor-list/>

484. <https://iabeurope.eu/all-news/update-on-the-belgian-data-protection-authoritys-investigation-of-iab-europe/>

came into play in California, IAB Tech Lab US developed the U.S. Privacy<sup>485</sup> (USP) technical specifications, using the same concepts.

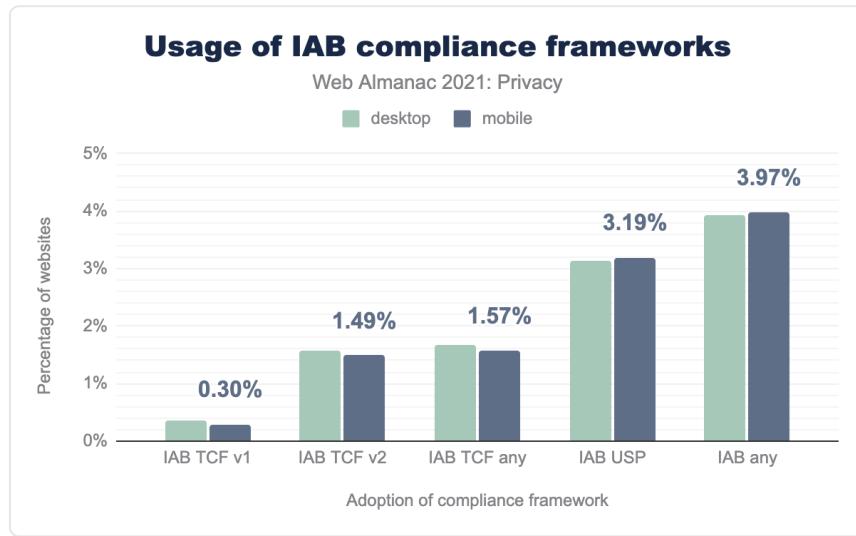


Figure 11.23. Percentage of websites using IAB compliance frameworks.

Above, we show the distribution of the usage of both versions of TCF and of USP. Note that the crawl is US-based, therefore we do not expect many websites to have implemented TCF. Fewer than 2% of websites use any TCF version, while twice as many websites use the US Privacy framework.

<sup>485</sup> <https://iabtechlab.com/standards/ccpa/>

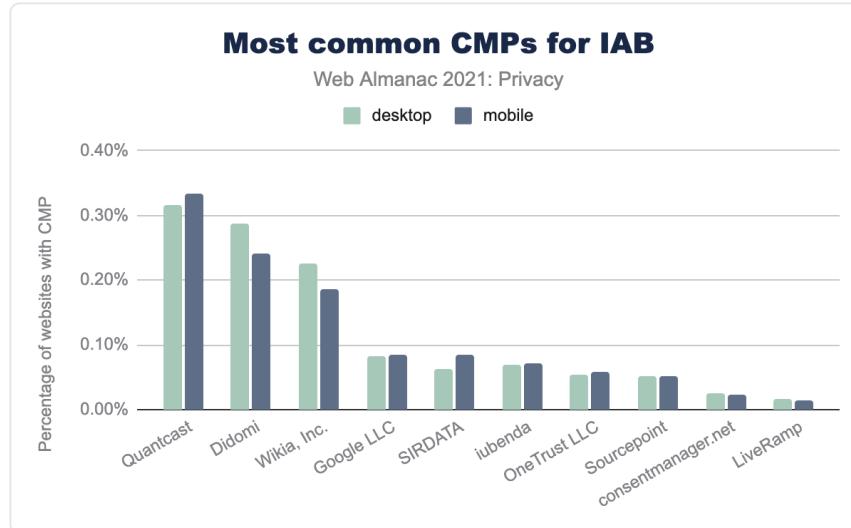


Figure 11.24. 10 most popular consent management platforms for IAB.

In the 10 most popular consent management platforms that are part of the framework, at the top we find Quantcast<sup>486</sup> with 0.34% on mobile. Other popular solutions are Didomi<sup>487</sup> with 0.24%, and Wikia, with 0.30%.

In the USP framework, the website's and user's privacy settings are encoded in a *privacy string*.

486. <https://www.quantcast.com/products/choice-consent-management-platform/>

487. <https://www.didomi.io/>



Figure 11.25. Percentage of websites using IAB US privacy strings.

The most common privacy string is `1---`. This indicates that CCPA does not apply to the website and therefore the website is not obliged to provide an opt-out for the user. CCPA only applies to companies whose main business involves selling personal data, or to companies that process data and have an annual turnover of more than \$25 million. The second most recurring string is `1YNY`. This indicates that the website provided “notice and opportunity to opt-out of sale of data”, but that the user has *not* opted out of the sale of their personal data.

## Privacy policies

Nowadays, most websites have a privacy policy, where users can learn about the types of information that is stored and processed about them.

**39.70%**

Figure 11.26. Percentage of mobile websites with a privacy policy link.

By looking for keywords such as “privacy policy”, “cookie policy”, and more, in a number of

languages<sup>488</sup>, we see that 39.70% of mobile websites, and 43.02% of desktop sites refer to some sort of privacy policy. While some websites are not required to have such a policy, many websites handle personal data and should therefore have a privacy policy to be fully transparent towards their users.

## Do Not Track - Global Privacy Control

The Do Not Track<sup>489</sup> (DNT) HTTP header can be used to communicate to websites that a user does not wish to be tracked. We can see the number of sites that appear to access the current value for DNT below, based on the presence of the `Navigator.doNotTrack` JavaScript call.

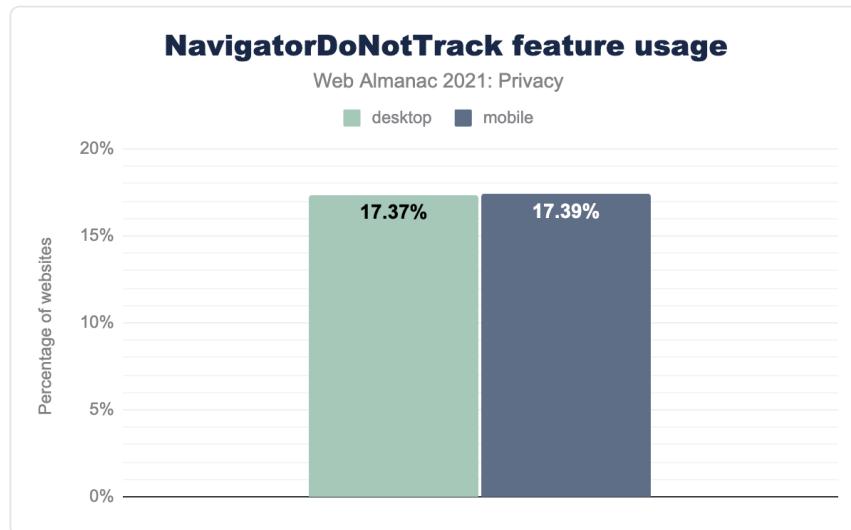


Figure 11.27. Percentage of websites using Do Not Track (DNT).

Around the same percentage of pages on mobile and desktop clients use DNT. However, in practice hardly any websites actually respect the DNT opt-outs. The Tracking Protection Working Group, which specifies DNT, closed down<sup>490</sup> in 2018, due to “lack of support”<sup>491</sup>. Safari then stopped supporting DNT<sup>492</sup> to prevent potential abuse for fingerprinting.

DNT’s successor Global Privacy Control<sup>493</sup> (GPC) was released in October 2020 and is meant to provide a more enforceable alternative, with the hopes of better adoption. This privacy

488. [https://github.com/RUB-SysSec/we-value-your-privacy/blob/master/privacy\\_wording.json](https://github.com/RUB-SysSec/we-value-your-privacy/blob/master/privacy_wording.json)

489. <https://www.w3.org/Issues/do-not-track>

490. <https://www.w3.org/2016/11/tracking-protection-wg.html>

491. <https://lists.w3.org/Archives/Public/public-tracking/2018Oct/0000.html>

492. [https://developer.apple.com/documentation/safari-release-notes/safari-12\\_1-release-notes#-text-Removed%20support%20for%20the%20expired%20Do%20Not%20Track](https://developer.apple.com/documentation/safari-release-notes/safari-12_1-release-notes#-text-Removed%20support%20for%20the%20expired%20Do%20Not%20Track)

493. <https://globalprivacycontrol.org/>

preference signal is implemented with a single bit in all HTTP requests. We did not yet observe any uptake, but we can expect this to improve in future as major browsers are now starting to implement GPC<sup>494</sup>.

## How browsers are evolving their privacy approaches

Given the push to better protect users' privacy while browsing the web, major browsers are implementing new features that should better safeguard users' sensitive data. We already covered ways in which browsers have started enforcing more privacy-preserving default settings for `Referrer-Policy` headers and `SameSite` cookies.

Furthermore, Firefox and Safari seek to block tracking through Enhanced Tracking Protection<sup>495</sup> and Intelligent Tracking Prevention<sup>496</sup> respectively.

Beyond blocking trackers, Chrome has launched the Privacy Sandbox<sup>497</sup> to develop new web standards that provide more privacy-friendly functionality for various use cases, such as advertising and fraud protection. We'll look more closely at these up-and-coming technologies that are designed to reduce the opportunity for sites to track users.

### Privacy Sandbox

To seek ecosystem feedback, early and experimental versions of Privacy Sandbox APIs are made available initially behind feature flags<sup>498</sup> for testing by individual developers, and then in Chrome via *origin trials*. Sites can take part in these origin trials to test experimental web platform features, and give feedback to the web standards community on a feature's usability, practicality, and effectiveness, before it's made available to all websites by default.

***Disclaimer:** Origin trials are only available for a limited amount of time. The numbers below represent the state of Privacy Sandbox origin trials at the time of this writing, in October 2021.*

### FLoC

One of the most hotly debated Privacy Sandbox experiments has been *Federated Learning of Cohorts*, or FLoC for short. The origin trial for FLoC ended in July 2021.

Interest-based ad selection is commonly used on the web. FLoC provided an API to meet that

494. <https://www.washingtonpost.com/technology/2021/10/26/global-privacy-control-firefox/>

495. [https://developer.mozilla.org/docs/Web/Privacy/Tracking\\_Protection](https://developer.mozilla.org/docs/Web/Privacy/Tracking_Protection)

496. <https://webkit.org/tracking-prevention/>

497. <https://privacysandbox.com/>

498. <https://www.chromium.org/developers/how-tos/run-chromium-with-flags>

specific use case without the need to identify and track individual users. FLoC has taken some flak<sup>499</sup>: Firefox<sup>500</sup> and other Chromium-based browsers<sup>501</sup> have declined to implement it, and the Electronic Frontier Foundation has voiced concerns that it might introduce new privacy risks<sup>502</sup>. However, FLoC was a first experiment. Future iterations of the API could alleviate these concerns and see wider adoption.

With FLoC, instead of assigning unique identifiers to users, the browser determined a user's *cohort*: a group of thousands of people who visited similar pages and may therefore be of interest to the same advertisers.

Since FLoC was an experiment, it was not widely deployed. Instead, websites could test it by enrolling in an origin trial. We found 62 and 64 websites that tested FLoC across desktop and mobile respectively.

Here is how the first FLoC experiment worked: as a user moved around the web, their browser used the FLoC algorithm to work out its *interest cohort*, which was the same for thousands of browsers with a similar recent browsing history. The browser recalculated its cohort periodically, on the user's device, without sharing individual browsing data with the browser vendor or other parties. When working out its cohort, a browser was choosing between cohorts that didn't reveal sensitive categories<sup>503</sup>.

Individual users and websites could opt out of being included in the cohort calculation.

499. <https://www.economist.com/the-economist-explains/2021/05/17/why-is-floc-googles-new-ad-technology-taking-flak>

500. <https://blog.mozilla.org/en/privacy-security/privacy-analysis-of-floc/>

501. <https://www.theverge.com/2021/4/16/22387492/google-floc-ad-tech-privacy-browsers-brave-vivaldi-edge-mozilla-chrome-safari>

502. <https://www.eff.org/deeplinks/2021/03/googles-floc-terrible-idea>

503. <https://www.chromium.org/Home/chromium-privacy/privacy-sandbox/floc#:~:text=web%20pages%20on%20sensitive%20topics>



Figure 11.28. Percentages of websites that opt out of FLoC cohorts.

We saw that 4.10% of the top 1,000 websites have opted out of FLoC. Across all websites, under 1% have opted out.

## Other Privacy Sandbox experiments

Within Google's Privacy Sandbox initiative, a number of experiments are in various stages of development.

The *Attribution Reporting API* (previously called *Conversion Measurement*) makes it possible to measure when user interaction with an ad leads to a conversion—for example, when an ad click eventually led to a purchase. We saw the first origin trial (which ended in October 2021) enabled on 10 origins.

*FLEDGE* (First “Locally-Executed Decision over Groups” Experiment) seeks to address ad targeting. The API can be tested in current versions of Chrome locally by individual developers<sup>504</sup> but there is no origin trial as of October 2021.

*Trust Tokens* enable a website to convey a limited amount of information from one browsing context to another to help combat fraud, without passive tracking. We saw the first origin trial<sup>505</sup> (which will end in May 2022) enabled on 7 origins that are likely embedded in a number of sites as third-party providers.

504. <https://developer.chrome.com/docs/privacy-sandbox/fledge/>

505. <https://developer.chrome.com/blog/third-party-origin-trials/>

**CHIPS** (Cookies Having Independent Partitioned State) allows websites to mark cross-site cookies as “Partitioned”, putting them in a separate cookie jar per top-level site. (Firefox has already introduced the similar *Total Cookie Protection* feature for cookie partitioning.) As of October 2021, there is no origin trial for CHIPS.

**Fenced Frames** protect frame access to data from the embedding page. As of October 2021, there is no origin trial.



Figure 11.29. Percentage of cookies with the `SameParty` cookie attribute.

Finally, *First-Party Sets* allow website owners to define a set of distinct domains that actually belong to the same entity. Owners can then set a `SameParty` attribute on cookies that should be sent across cross-site contexts, as long as the sites are in the same first-party set. A first origin trial ended in September 2021. We saw the `SameParty` attribute on a few thousand cookies.

## Conclusion

Users’ privacy remains at risk on the web today: over 80% of all websites have some form of tracking enabled, and novel tracking mechanisms such as CNAME tracking are being developed. Some sites also handle sensitive data such as geolocation, and if they’re not careful, potential breaches could result in users’ personal data being exposed.

Fortunately, increased awareness about the need for privacy on the web has led to concrete

action. Websites now have access to features that allow them to safeguard access to sensitive resources. Legislation across the globe enforces explicit user consent for sharing personal data. Websites are implementing privacy policies and cookie banners to comply. Finally, browsers are proposing and developing innovative technologies to continue supporting use cases such as advertising and fraud detection in a more privacy-friendly way.

Ultimately, users should be empowered to have a say in how their personal data is treated. Meanwhile, browsers and website owners should develop and deploy the technical means to guarantee that users' privacy is protected. By incorporating privacy throughout our interactions with the web, users can feel more certain that their personal data is well protected.

## Authors



**Yana Dimova**

ydimova

Yana Dimova is a PhD student at imec-DistriNet, working on web privacy. Her general interests and work focus on online tracking, privacy vulnerabilities and privacy legislation and policies.



**Victor Le Pochat**

@VictorLePochat VictorLeP victor-le-pochat https://lepoch.at

Victor Le Pochat is a PhD researcher at the imec-DistriNet<sup>506</sup> research group of KU Leuven in Belgium. His interests lie in the exploration of web ecosystems, and in web security/privacy research methodology, both analyzing and improving current methods.

---

506. <https://distriinet.cs.kuleuven.be/>



# Part II Chapter 12

# Security



*Written by Saptak Sengupta, Tom Van Goethem, and Nurullah Demir*

*Reviewed by Caleb Queern, Edmond W. W. Chan, and Matteo Große-Kampmann*

*Analyzed by Gertjan Franken*

*Edited by Barry Pollard*

## Introduction

We are becoming more and more digital today. We are not only digitizing our business but also our private life. We contact people online, send messages, share moments with friends, do our business, and organize our daily routine. At the same time, this shift means that more and more critical data is being digitized and processed privately and commercially. In this context, cybersecurity is also becoming more and more important as its goal is to safeguard users by offering availability, integrity and confidentiality of user data. When we look at today's technology, we see that web resources are increasingly used to provide digitally delivered solutions. It also means that there is a strong link between our modern life and the security of web applications due to their widespread use.

This chapter analyzes the current state of security on the web and gives an overview of methods that the web community uses (and misses) to protect their environment. More specifically, in this report, we analyze different metrics on Transport Layer Security (HTTPS), such as general implementation, protocol versions, and cipher suites. We also give an overview

of the techniques used to protect cookies. You will then find a comprehensive analysis on the topic of content inclusion and methods for thwarting attacks (e.g., use of specific security headers). We also look at how the security mechanisms are adopted (e.g., by country or specific technology). We also discuss malpractices on the web, such as Cryptojacking and, finally we look at usage of `security.txt` URLs.

We crawl the analyzed pages in both desktop and mobile mode, but for a lot of the data they give similar results, so unless otherwise noted, stats presented in this chapter refer to the set of mobile pages. For more information on how the data has been collected, refer to the Methodology page.

## Transport security

Following the recent trend, we see continuous growth in the number of websites adopting HTTPS this year as well. Transport Layer Security is important to allow secure browsing of websites by ensuring that the resources being served to you and the data sent to the website are untampered in the transit. Almost all major browsers now come with a HTTPS-only setting and increasing warnings are shown to users when HTTP is used by a website instead of HTTPS, thus pushing broader adoption forward.

A large, bold, blue percentage value '91.1%' is displayed, likely representing the percentage of mobile requests using HTTPS.

*Figure 12.1. The percentage of requests that use HTTPS on mobile.*

Currently, we see that 91.9% of total requests for websites on desktop and 91.1% for mobile are being served using HTTPS. We see an increasing number of certificates<sup>507</sup> being issued every day thanks to non-profit certificate authorities like Let's Encrypt.

507. <https://letsencrypt.org/stats/#daily-issuance>

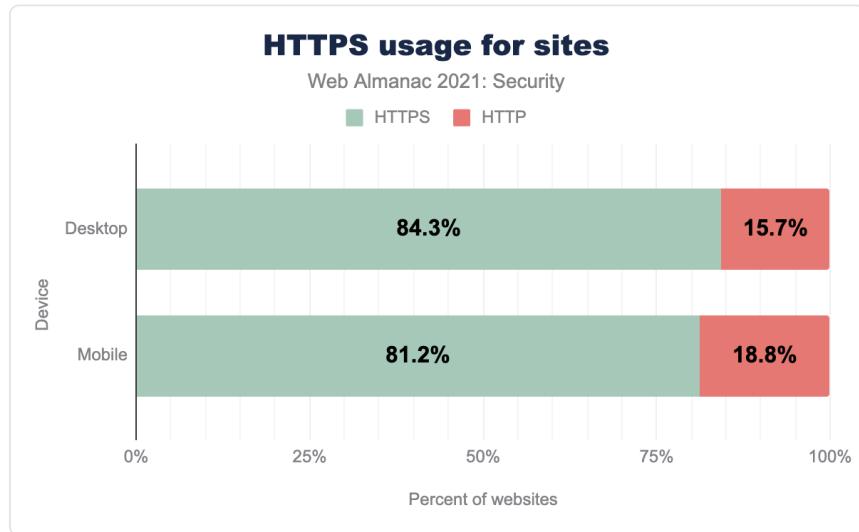


Figure 12.2. HTTPS usage for sites.

Currently, 84.3% of website homepages in desktop and 81.2% of website homepages in mobile are served over HTTPS so we still see a gap between websites using HTTPS and requests using HTTPS. This is because a lot of the impressive percentage of HTTPS requests are often dominated by third-party services like fonts, analytics, CDNs, and not the initial web page itself.

We do see a continuous improvement in sites using HTTPS (approximately 7-8% increase since last year<sup>508</sup>), but soon a lot of unmaintained websites might start seeing warnings once browsers start adopting HTTPS-only mode by default<sup>509</sup>.

## Protocol versions

*Transport Layer Security (TLS)* is the protocol that helps make HTTP requests secure and private. With time, new vulnerabilities are discovered and fixed in TLS. Hence, it's not just important to serve a website over HTTPS but also to ensure that modern, up-to-date TLS configuration is being used to avoid such vulnerabilities.

As part of this effort to improve security and reliability by adopting modern versions, TLS 1.0 and 1.1 have been deprecated by the Internet Engineering Task Force (IETF)<sup>510</sup> as of March 25, 2021. All upstream browsers have also either completely removed support or deprecated TLS 1.0 and 1.1. For example, Firefox has deprecated TLS 1.0 and 1.1 but has not completely

508. <https://almanac.httparchive.org/en/2020/security/#fig-3>

509. <https://blog.mozilla.org/security/2021/08/10/firefox-91-introduces-https-by-default-in-private-browsing/>

removed it<sup>511</sup> because during the pandemic, users might need to access government websites that often still run on TLS 1.0. The user may still decide to change `security.tls.version.min` in browser config to decide the lowest TLS version they want the browser to allow.



Figure 12.3. TLS versions usage for sites.

60.4% of pages in desktop and 62.1% of pages in mobile are now using TLSv1.3, making it the majority protocol version over TLSv1.2. The number of pages using TLSv1.3 has increased approximately 20% since last year<sup>512</sup> when we saw 43.2% and 45.4% respectively.

## Cipher suites

Cipher suites are a set of algorithms that are used with TLS to help make secure connections. Modern Galois/Counter Mode<sup>513</sup> (GCM) cipher modes are considered to be much more secure compared to the older Cipher Block Chaining Mode<sup>514</sup> (CBC) ciphers which have shown to be vulnerable to padding attacks<sup>515</sup>. While TLSv1.2 did support use of both newer and older cipher suites, TLSv1.3 does not support any of the older cipher suites<sup>516</sup>. This is one reason TLSv1.3 is the more secure option for connections.

511. <https://www.ghacks.net/2020/03/21/mozilla-re-enables-tls-1-0-and-1-1-because-of-coronavirus-and-google/>

512. <https://almanac.httparchive.org/en/2020/security#protocol-versions>

513. [https://en.wikipedia.org/wiki/Galois/Counter\\_Mode](https://en.wikipedia.org/wiki/Galois/Counter_Mode)

514. [https://en.wikipedia.org/wiki/Block\\_cipher\\_mode\\_of\\_operation#Cipher\\_block\\_chaining\\_\(CBC\)](https://en.wikipedia.org/wiki/Block_cipher_mode_of_operation#Cipher_block_chaining_(CBC))

515. <https://blog.qualys.com/product-tech/2019/04/22/zombie-poodle-and-goldendoodle-vulnerabilities>

516. <https://datatracker.ietf.org/doc/html/rfc8446#page-133>

# 96.8%

Figure 12.4. Mobile sites using forward secrecy.

Almost all modern cipher suites support *Forward Secrecy* key exchange, meaning in the case that the server's keys are compromised, old traffic that used those keys cannot be decrypted. 96.6% in desktop and 96.8% in mobile use forward secrecy. TLSv1.3 has made forward secrecy compulsory though it is optional in TLSv1.2—yet another reason it is more secure.

The other consideration apart from the cipher mode is the key size of the Authenticated Encryption and Authenticated Decryption<sup>517</sup> algorithm. A larger key size will take a lot longer to compromise and the intensive computations for encryption and decryption of the connection impose little to no perceptible impact to site performance



Figure 12.5. Distribution of cipher suites.

**AES\_128\_GCM** is still the most widely used cipher suite, by a long way, with 79.4% in desktop and 78.9% in mobile usage. **AES\_128\_GCM** indicates that it uses GCM cipher mode with *Advanced Encryption Standard* (AES) of key size 128-bit for encryption and decryption. 128-bit key size is still considered secured, but 256-bit size is slowly becoming the industry standard to better resist brute force attacks for a longer time.

517. <https://datatracker.ietf.org/doc/html/rfc5116#section-2>

## Certificate Authorities

A **Certificate Authority** is a company or organization that issues digital certificates which helps validate the ownership and identity of entities on the web, like websites. A Certificate Authority is needed to issue a TLS certificate recognized by browsers so that the website can be served over HTTPS. Like the previous year, we will again look into the CAs used by websites themselves rather than third-party services and resources.

<b>Issuer</b>	<b>Algorithm</b>	<b>Desktop</b>	<b>Mobile</b>
R3 <sup>518</sup>	RSA	46.9%	49.2%
Cloudflare Inc ECC CA-3	ECDSA	11.7%	11.5%
Sectigo RSA Domain Validation Secure Server CA <sup>519</sup>	RSA	8.3%	8.2%
cPanel, Inc. Certification Authority	RSA	5.0%	5.5%
Go Daddy Secure Certificate Authority - G2 <sup>520</sup>	RSA	3.6%	3.0%
Amazon <sup>521</sup>	RSA	3.4%	3.0%
Encryption Everywhere DV TLS CA - G1 <sup>522</sup>	RSA	1.3%	1.6%
AlphaSSL CA - SHA256 - G2 <sup>523</sup>	RSA	1.2%	1.2%
RapidSSL TLS DV RSA Mixed SHA256 2020 CA-1 <sup>524</sup>	RSA	1.2%	1.1%
DigiCert SHA2 Secure Server CA <sup>525</sup>	RSA	1.1%	0.9%

Figure 12.6. Top 10 certificate issuers for websites.

Let's Encrypt has changed their subject common name<sup>526</sup> from "Let's Encrypt Authority X3" to just "R3" to save bytes in new certificates. So, any SSL certificates signed by R3 are issued by Let's Encrypt<sup>527</sup>. Thus, like previous years, we see Let's Encrypt continue to lead the charts with 46.9% of desktop websites and 49.2% of mobile sites using certificates issued by them. This is up 2-3% from last year. Its free, automated certificate generation has played a game-changing role in making it easier for everyone to serve their websites over HTTPS.

Cloudflare continues to be in second position with its similarly free certificates for its

518. <https://letsencrypt.org/certificates/>

519. <https://sectigo.com/knowledge-base/detail/Sectigo-Intermediate-Certificates/KA01N000000rfBO>

520. <https://certs.godaddy.com/repository>

521. <https://www.amazontrust.com/repository>

522. <https://www.digicert.com/kb/digicert-root-certificates.htm>

523. <https://support.globalsign.com/co-certificates/intermediate-certificates/alphassl-intermediate-certificates>

524. <https://www.digicert.com/kb/digicert-root-certificates.htm>

525. <https://www.digicert.com/kb/digicert-root-certificates.htm>

526. <https://letsencrypt.org/2020/09/17/new-root-and-intermediates.html#why-we-issued-an-ecdsa-root-and-intermediates>

527. <https://letsencrypt.org/certificates/>

customers. Also, Cloudflare CDNs increase the usage of *Elliptic Curve Cryptography* (ECC) certificates which are smaller and more efficient than RSA certificates but are often difficult to deploy, due to the need to also continue to serve non-ECC certificates to older clients. Using a CDN like Cloudflare takes care of that complexity for you. All the latest browsers<sup>528</sup> are compatible with ECC certificates, though some browsers like Chrome depend on the OS. So, if someone uses Chrome in an old OS like Windows XP, then they need to fall back to non-ECC certificates.

## HTTP Strict Transport Security

*HTTP Strict Transport Security* (HSTS) is a response header that tells the browser that it should always use secure HTTPS connections to communicate with the website.



Figure 12.7. The percentage of requests that have HSTS header on mobile.

The `Strict-Transport-Security` header helps convert a `http://` URL to a `https://` URL before a request is made for that site. 22.2% of the mobile responses and 23.9% of desktop responses have a HSTS header.

HSTS Directive	Desktop	Mobile
<code>Valid max-age</code>	92.7%	93.4%
<code>includeSubdomains</code>	34.5%	33.3%
<code>preload</code>	17.6%	18.0%

Figure 12.8. Usage of HSTS directives.

Out of the sites with HSTS header, 92.7% in desktop and 93.4% in mobile have a valid `max-age` (that is, the value is non-zero and non-empty) which determines how many seconds the browser should only visit the website over HTTPS.

33.3% of request responses for mobile, and 34.5% for desktop include `includeSubdomain` in the HSTS settings. The number of responses with the `preload` directive is lower because it is not part of the HSTS specification<sup>529</sup> and needs a minimum `max-age` of 31,536,000 seconds (or

528. <https://developers.cloudflare.com/ssl/ssl-tls/browser-compatibility>

529. [https://developer.mozilla.org/docs/Web/HTTP/Headers/Strict-Transport-Security#preloading\\_strict\\_transport\\_security](https://developer.mozilla.org/docs/Web/HTTP/Headers/Strict-Transport-Security#preloading_strict_transport_security)

1 year) and also the `includeSubdomain` directive to be present.



Figure 12.9. HSTS `max-age` values for all requests (in days).

The median value for `max-age` attribute in HSTS headers over all requests is 365 days in both mobile and desktop. <https://hstspreload.org/> recommends a `max-age` of 2 years once the HSTS header is set up properly and verified to not cause any issues.

## Cookies

An *HTTP cookie* is a small piece of information about the user accessing the website that the server sends to the web browser. Browsers store this information and send it back with subsequent requests to the server. Cookies help in session management to maintain state information of the user, such as if the user is currently logged in.

Without properly securing cookies, an attacker can hijack a session and send unwanted changes to the server by impersonating the user. It can also lead to *Cross-Site Request Forgery* attacks, whereby the user's browser inadvertently sends a request, including the cookies, unbeknownst to the user.

Several other types of attacks rely on the inclusion of cookies in cross-site requests, such as *Cross-Site Script Inclusion (XSSI)* and various techniques in the *XS-Leaks* vulnerability class.

You can ensure that cookies are sent securely and aren't accessed by unintended parties or

scripts by adding certain attributes or prefixes.

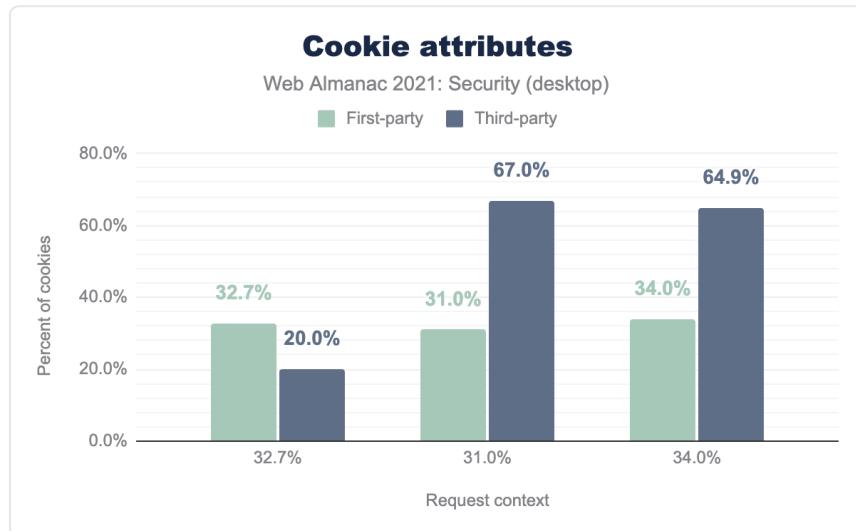


Figure 12.10. Cookie attributes (desktop).

## Secure

Cookies that have the `Secure` attribute set will only be sent over a secure HTTPS connection, preventing them from being stolen in a *Manipulator-in-the-middle* attack. Similar to HSTS, this also helps enhance the security provided by TLS protocols. For first-party cookies, just over 30% of the cookies in both desktop and mobile have the `Secure` attribute set. However, we do see a significant increase in the percentage of third-party cookies in desktop having the `Secure` attribute from 35.2% last year<sup>530</sup> to 67.0% this year. This increase is likely due to the `Secure` attribute being a requirement for `SameSite=none` cookies, that we will discuss below.

## HttpOnly

A cookie that has the `HttpOnly` attribute set cannot be accessed through the `document.cookie` API in JavaScript. Such cookies can only be sent to the server and helps in mitigating client-side Cross-Site Scripting (XSS) attacks that misuse the cookie. It's used for cookies that are only needed for server-side sessions. The percentage of cookies with

530. <https://almanac.httparchive.org/en/2020/security#cookies>

`HttpOnly` attribute has a smaller difference between first-party cookies and third-party compared to the other cookie attributes being used by 32.7% and 20.0% respectively.

## SameSite

The `SameSite` attribute in cookies allows the websites to inform the browser when and whether to send a cookie with cross-site requests. This is used to prevent cross-site request forgery attacks. `SameSite=Strict` allows the cookie to be sent only to the site where it originated. With `SameSite=Lax`, cookies are not sent to cross-site requests unless a user is navigating to the origin site by following a link. `SameSite=None` means cookies are sent in both originating and cross-site requests.

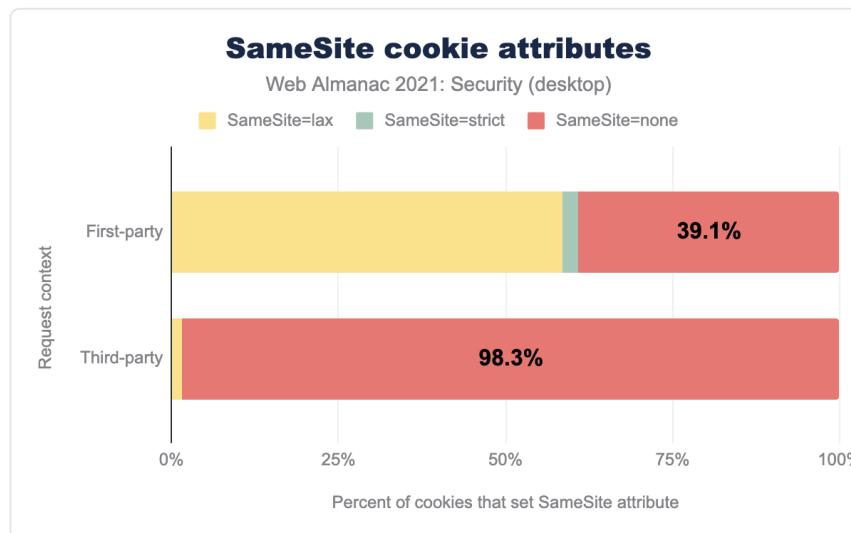


Figure 12.11. Same site cookie attributes.

We see that 58.5% of all first-party cookies with a `SameSite` attribute have the attribute set to `Lax` while there is still a pretty daunting 39.1% cookies where `SameSite` attribute is set to `none`—although the number is steadily decreasing. Almost all current browsers now default to `SameSite=Lax` if no `SameSite` attribute is set. Approximately 65% of overall first-party cookies have no `SameSite` attribute.

## Prefixes

Cookie prefixes `_Host-` and `_Secure-` help mitigate attacks to override the session

cookie information for a session fixation attack<sup>531</sup>. `_Host`- helps in domain locking a cookie by requiring the cookie to also have `Secure` attribute, `Path` attribute set to `/`, not have `Domain` attribute and to be sent from a secure origin. `_Secure`- on the other hand requires the cookie to only have `Secure` attribute and to be sent from a secure origin.

Type of cookie	<code>_Secure</code>	<code>_Host</code>
First-party	0.02%	0.01%
Third-party	< 0.01%	0.03%

Figure 12.11. Usage of `_Secure` and `_Host` cookie prefixes in mobile.

Though both the prefixes are used in a significantly lower percentage of cookies, `_Secure`- is more commonly found in first-party cookies due to its lower prerequisites.

## Cookie age

Permanent cookies are deleted at a date specified by the `Expires` attribute, or after a period of time specified by the `Max-Age` attribute. If both `Expires` and `Max-Age` are set, `Max-Age` has precedence.

531. [https://owasp.org/www-community/attacks/Session\\_fixation](https://owasp.org/www-community/attacks/Session_fixation)



Figure 12.12. Cookie age usage in days (mobile).

We see that the median `Max-Age` is 365 days, as we see about 20.5% of the cookies with `Max-Age` have the value 31,536,000. However, 64.2% of the first-party cookies have `Expires` and 23.3% have `Max-Age`. Since `Expires` is much more dominant among cookies, the median for real maximum age is the same as `Expires` (180 days) instead of `Max-Age` as you would expect.

## Content inclusion

Most websites have quite a lot of media and CSS or JavaScript libraries that more often than not are loaded from various different external sources, CDNs or cloud storage services. It's important for the security of the website as well as the security of the users of a website to ensure which source of content can be trusted. Otherwise, the website is vulnerable to cross-site scripting attacks if untrusted content gets loaded.

## Content Security Policy

**Content Security Policy (CSP)** is the predominant method used to mitigate cross-site scripting and data injection attacks by restricting the origins allowed to load various content. There are numerous directives that can be used by the website to specify sources for different kinds of content. For instance, `script-src` is used to specify origins or domains from which scripts can be loaded. It also has other values to define if inline scripts and `eval()` functions are

allowed.

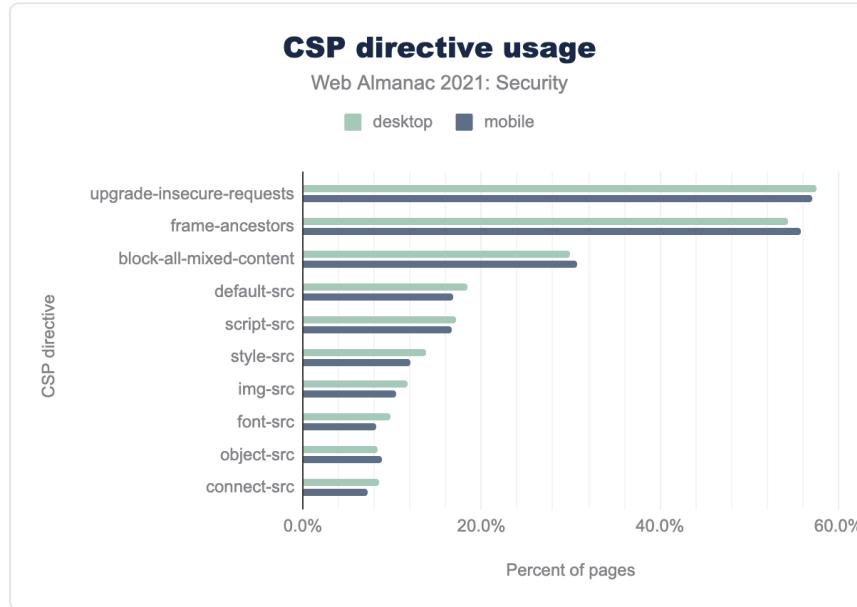


Figure 12.13. Most common directives used in CSP.

We see more and more websites starting to use CSP with 9.3% homepages on mobile using CSP now compared to 7.2% last year. `upgrade-insecure-requests` continues to be the most frequent CSP used. The high adoption rate for this policy is likely because of the same reasons mentioned last year<sup>532</sup>; it is an easy, low-risk, policy that helps in upgrading all HTTP requests to HTTPS and also helps with to block mixed content being used on the page. `frame-ancestors` is a close second, which helps one define valid parents that may embed a page.

The adoption of policies defining the sources from which content can be loaded continues to be low. Most of these policies are more difficult to implement, as they can cause breakages. They require effort to implement to define `nonce`, hashes or domains for allowing external content.

While a strict CSP is a strong defense against attacks, they can lead to undesirable effects and prevent valid content from loading, if the policy is incorrectly defined. Different libraries and APIs loading further content makes this even more difficult.

Lighthouse<sup>533</sup> recently started flagging severity warnings when such directives are missing from CSP, encouraging people to adopt a stricter CSP to prevent XSS attacks. We will discuss more

532. <https://almanac.httparchive.org/en/2020/security#content-security-policy>

533. <https://web.dev/csp-xss/>

about how CSP helps in stopping XSS attacks in the thwarting attacks section of this chapter.

To allow web developers to evaluate the correctness of their CSP policy, there is also 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 still fairly small: 0.9% in mobile. However, most of the time this header is added in the testing phase and later is replaced by the enforcing CSP, so the low usage is not unexpected.

Sites can also use the `report-uri` directive to report any CSP violations to a particular link that is able to parse the CSP errors. These can help after a CSP directive has been added to check if any valid content is accidentally being blocked by the new directive. The drawback of this powerful feedback mechanism is that CSP reporting can be noisy due to browser extensions and other technology outside of the website owner's control.

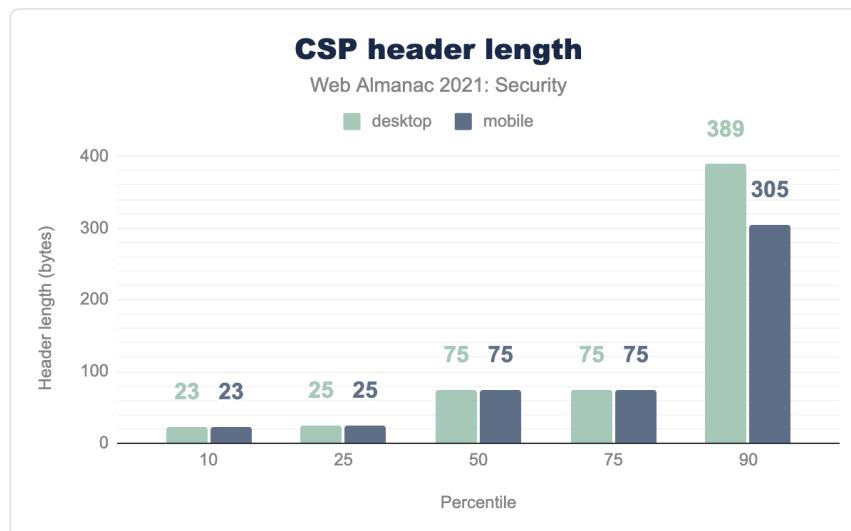


Figure 12.14. CSP header length.

The median length of CSP headers continue to be pretty low: 75 bytes. Most websites still use single directives for specific purposes, instead of long strict CSPs. For instance, 24.2% of websites only have `upgrade-insecure-requests` directives.

# 43,488

Figure 12.15. Bytes in the longest CSP observed.

On the other side of the spectrum, the longest CSP header is almost twice as long as last year's longest CSP header: 43,488 bytes.

<b>Origin</b>	<b>Desktop</b>	<b>Mobile</b>
<code>https://www.google-analytics.com</code>	0.29%	0.22%
<code>https://www.googletagmanager.com</code>	0.26%	0.22%
<code>https://fonts.googleapis.com</code>	0.22%	0.16%
<code>https://fonts.gstatic.com</code>	0.20%	0.15%
<code>https://www.google.com</code>	0.19%	0.14%
<code>https://www.youtube.com</code>	0.19%	0.13%
<code>https://connect.facebook.net</code>	0.16%	0.11%
<code>https://stats.g.doubleclick.net</code>	0.15%	0.11%
<code>https://www.gstatic.com</code>	0.14%	0.11%
<code>https://cdnjs.cloudflare.com</code>	0.12%	0.10%

Figure 12.16. Most frequently allowed hosts in CSP policies.

The most common origins used in `* -src` directives continue to be heavily dominated by Google (fonts, ads, analytics). We also see Cloudflare's popular library CDN showing up in the 10th position this year.

## Subresource Integrity

A lot of websites, load JavaScript libraries and CSS libraries from external CDNs. This can have certain security implications if the CDN is compromised, or an attacker finds some other way to replace the frequently used libraries. *Subresource Integrity* (SRI) helps in avoiding such consequences, though it introduces other risks if the website may not function without that resource for a non-malicious change. Self-hosting instead of loading from a third party is usually a safer option where possible.

# 66.2%

Figure 12.17. Usage of SHA384 hash function for SRI in mobile.

Web developers can add the `integrity` attribute to `<script>` and `<link>` tags which are used to include JavaScript and CSS code to the website. The `integrity` attribute consists of a hash of the expected content of the resource. The browser can then compare the hash of the fetched content and hash mentioned in the `integrity` attribute to check its validity and only render the resource if they match.

```
<script src="https://code.jquery.com/jquery-3.6.0.min.js"
       integrity="sha256-/xUj+3OJU5yExlq6GSYGSKh7tPXikynS7ogEvDej/m4="
       crossorigin="anonymous"></script>
```

The hash can be computed with three different algorithms: `SHA256`, `SHA384`, and `SHA512`. `SHA384` (66.2% in mobile) is currently the most used, followed by `SHA256` (31.1% in mobile). Currently, all three hashing algorithms are considered safe to use.

A large, bold, blue percentage value '82.6%' centered on the page.

Figure 12.18. Percentage of SRI in `<script>` elements for mobile.

There has been some increase in the usage of SRI over the past couple of years, with 17.5% elements in desktop and 16.1% elements in mobile containing the `integrity` attribute. 82.6% of those were in the `<script>` element for mobile.



Figure 12.19. Subresource integrity: coverage per page.

However, it still is a minority option for `<script>` elements. The median percentage of `<script>` elements on websites which have an `integrity` attribute is 3.3%.

Host	Desktop	Mobile
www.gstatic.com	44.3%	44.1%
cdn.shopify.com	23.4%	23.9%
code.jquery.com	7.5%	7.5%
cdnjs.cloudflare.com	7.2%	6.9%
stackpath.bootstrapcdncdn.com	2.7%	2.7%
maxcdn.bootstrapcdncdn.com	2.2%	2.3%
cdn.jsdelivr.net	2.1%	2.1%

Figure 12.20. Most common hosts from which SRI-protected scripts are included.

Among the common hosts from which SRI-protected scripts are included, we see most of them are made up of CDNs. We see that there are three very common CDNs that are used by

multiple websites when using different libraries: jQuery<sup>534</sup>, cdnjs<sup>535</sup>, and Bootstrap<sup>536</sup>. It is probably not coincidental that all three of these CDNs have the integrity attribute in their example HTML code, so when developers use the examples to embed these libraries, they are ensuring that SRI-protected scripts are being loaded.

## Permissions Policy

All browsers these days provide a myriad of APIs and functionalities, which can be used for tracking and malicious purposes, thus proving detrimental to the privacy of the users.

*Permissions Policy* is a web platform API that gives a website the ability to allow or block the use of browser features in its own frame or in iframes that it embeds.

The `Permissions-Policy` response header allows websites to decide which features they want to use and also which powerful features they want to disallow on the website to limit misuse. A Permissions Policy can be used to control APIs like Geolocation, User media, Video autoplay, Encrypted media decoding and many more. While some of these APIs do require browser permission from the user—a malicious script can't turn on the microphone without the user getting a permission pop up—it's still good practice to use Permission Policy to restrict usage of certain features completely if they are not required by the website.

This API specification was previously known as *Feature Policy* but as well as the rename there have been many other updates. Though the `Feature-Policy` response header is still in use, it is pretty low with only 0.6% of websites in mobile using it. The `Permissions-Policy` response headers contains an allow list for different APIs. For example, `Permissions-Policy: geolocation=(self "https://example.com")` means that the website disallows the use of Geolocation API except for its own origin and those whose origin is "`https://example.com`". One can disable the use of an API entirely in a website by specifying an empty list, e.g., `Permissions-Policy: geolocation=()`.

We see 1.3% of websites on the mobile using the `Permissions-Policy` already. A possible reason for this higher than expected usage of this new header, could be some website admins choosing to opt-out of Federated Learning of Cohorts or FLoC<sup>537</sup> (which was experimentally implemented in Chrome) to protect user's privacy. The privacy chapter has a detailed analysis of this.

534. <https://code.jquery.com/>

535. <https://cdnjs.com/>

536. <https://www.bootstrapcdncdn.com/>

537. <https://privacysandbox.com/proposals/floc>

<b>Directive</b>	<b>Desktop</b>	<b>Mobile</b>
<code>encrypted-media</code>	46.8%	45.0%
<code>conversion-measurement</code>	39.5%	36.1%
<code>autoplay</code>	30.5%	30.1%
<code>picture-in-picture</code>	17.8%	17.2%
<code>accelerometer</code>	16.4%	16.0%
<code>gyroscope</code>	16.4%	16.0%
<code>clipboard-write</code>	11.2%	10.9%
<code>microphone</code>	4.3%	4.5%
<code>camera</code>	4.2%	4.4%
<code>geolocation</code>	4.0%	4.3%

Figure 12.21. Prevalence of `allow` directives on frames.

One can also use the `allow` attribute in `<iframe>` elements to enable or disable features allowed to be used in the embedded frame. 28.4% of 10.8 million frames in mobile contained the `allow` attribute to enable permission or feature policies.

As in previous years, the most used directives in `allow` attributes on iframes are still related to controls for embedded videos and media. The most used directive continues to be `encrypted-media` which is used to control access to the Encrypted Media Extensions API.

## Iframe sandbox

An untrusted third-party in an iframe could launch a number of attacks on the page. For instance, it could navigate the top page to a phishing page, launch popups with fake anti-virus advertisements and other cross-frame scripting attacks.

The `sandbox` attribute on iframes applies restrictions to the content, and therefore reduces the opportunities for launching attacks from the embedded web page. The value of the attribute can either be empty to apply all restrictions (the embedded page cannot execute any JavaScript code, no forms can be submitted, and no popups can be created, to name a few restrictions), or space-separated tokens to lift particular restrictions. As embedding third-party content such as advertisements or videos via iframes is common practice on the web, it is not

surprising that many of these are restricted via the `sandbox` attribute: 32.6% of the iframes on desktop pages have a `sandbox` attribute while on mobile pages this is 32.6%.



Figure 12.22. Prevalence of sandbox directives on frames.

The most commonly used directive, `allow-scripts`, which is present in 99.98% 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, for example, access cookies that were set on that origin.

## Thwarting attacks

Web applications can be vulnerable to multiple attacks. Fortunately, there exist several mechanisms that can either prevent certain classes of vulnerabilities (e.g., framing protection through `X-Frame-Options` or CSP's `frame-ancestors` directive) is necessary to combat clickjacking attacks<sup>538</sup>), or limit the consequences of an attack. As most of these protections are opt-in, they still need to be enabled by the web developers—typically by setting the correct response header. At large scale, the presence of the headers can tell us something about the security hygiene of websites and the incentives of the developers to protect their users.

538. <https://pragmaticwebsecurity.com/articles/securitypolicies/preventing-framing-with-policies.html>

## Security feature adoption

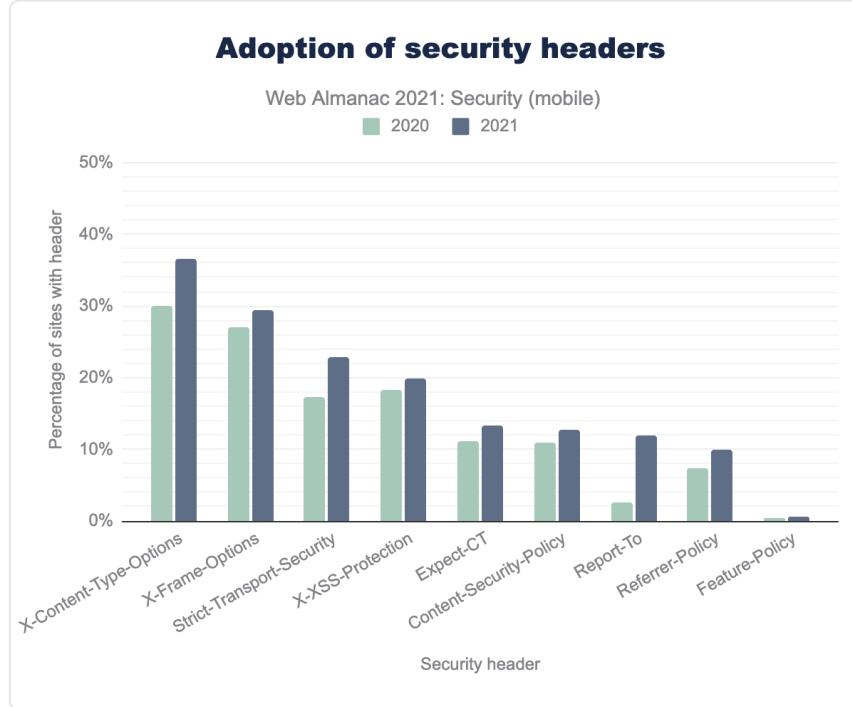


Figure 12.23. Adoption of security headers for site requests in mobile pages.

Perhaps the most promising and uplifting finding of this chapter is that the general adoption of security mechanisms continues to grow. Not only does this mean that attackers will have a more difficult time exploiting certain websites, but it is also indicative that more and more developers value the security of the web products they build. Overall, we can see a relative increase in the adoption of security features of 10-30% compared to last year. The security-related mechanism with the most uptake is the `Report-To` header of the Reporting API<sup>539</sup>, with almost a 4x increased adoption rate, from 2.6% to 12.2%.

Although this continued increase in the adoption rate of security mechanisms is certainly outstanding, there still remains quite some room for improvement. The most widely used security mechanism is still the `X-Content-Type-Options` header, which is used on 36.6% of the websites we crawled on mobile, to protect against MIME-sniffing attacks. This header is followed by the `X-Frame-Options` header, which is enabled on 29.4% of all sites.

Interestingly, only 5.6% of websites use the more flexible `frame-ancestors` directive of CSP.

<sup>539</sup>. <https://developers.google.com/web/updates/2018/09/reportingapi>

Another interesting evolution is that of the `X-XSS-Protection` header. The feature is used to control the XSS filter of legacy browsers: Edge<sup>540</sup> and Chrome<sup>541</sup> retired their XSS filter in July 2018 and August 2019 respectively as it could introduce new unintended vulnerabilities. Yet, we found that the `X-XSS-Protection` header was 8.5% more prevalent than last year.

## Features enabled in `<meta>` element

In addition to sending a response header, some security features can be enabled in the HTML response body by including a `<meta>` element with the `name` attribute set to `http-equiv`. For security purposes, only a limited number of policies can be enabled this way. More precisely, only a Content Security Policy and Referrer Policy can be set via the `<meta>` tag. Respectively we found that 0.4% and 2.6% of the mobile sites enabled the mechanism this way.

# 3,410

Figure 12.24. Number of sites with `X-Frame-Options` in the `<meta>` tag, which is actually ignored by the browser.

When any of the other security mechanisms are set via the `<meta>` tag, the browser will actually ignore this. Interestingly, we found 3,410 sites that tried to enable `X-Frame-Options` via a `<meta>` tag, and thus were wrongly under the impression that they were protected from clickjacking attacks. Similarly, several hundred websites failed to deploy a security feature by placing it in a `<meta>` tag instead of a response header (`X-Content-Type-Options`: 357, `X-XSS-Protection`: 331, `Strict-Transport-Security`: 183).

## Stopping XSS attacks via CSP

CSP can be used to protect against a multitude of things: clickjacking attacks, preventing mixed-content inclusion and determining the trusted sources from which content may be included (as discussed above).

Additionally, it is an essential mechanism to defend against XSS attacks. For instance, by setting a restrictive `script-src` directive, a web developer can ensure that only the application's JavaScript code is executed (and not the attacker's). Moreover, to defend against DOM-based cross-site scripting, it is possible to use *Trusted Types*, which can be enabled by using CSP's `require-trusted-types-for` directive.

540. <https://blogs.windows.com/windows-insider/2018/07/25/announcing-windows-10-insider-preview-build-17723-and-build-18204/>

541. <https://www.chromium.org/developers/design-documents/xss-auditor>

<b>Keyword</b>	<b>Desktop</b>	<b>Mobile</b>
<code>strict-dynamic</code>	5.2%	4.5%
<code>nonce-</code>	12.1%	17.6%
<code>unsafe-inline</code>	96.2%	96.5%
<code>unsafe-eval</code>	82.9%	77.2%

Figure 12.25. Prevalence of CSP keywords based on policies that define a `default-src` or `script-src` directive.

Although we saw an overall moderate increase (17%) in the adoption of CSP, what is perhaps even more exciting is that the usage of the `strict-dynamic` and nonces is either keeping the same trend or is slightly increasing. For instance, for desktop sites the use of `strict-dynamic` grew from 2.4% last year<sup>542</sup>, to 5.2% this year. Similarly, the use of nonces grew from 8.7% to 12.1%.

On the other hand, we find that the usage of the troubling directives `unsafe-inline` and `unsafe-eval` is still fairly high. However, it should be noted that if these are used in conjunction with `strict-dynamic`, modern browsers will ignore these values, while older browsers without `strict-dynamic` support can still continue to use the website.

## Defending against XS-Leaks

Various new security features have been introduced to allow web developers to defend their websites against micro-architectural attacks, such as Spectre<sup>543</sup>, and other attacks that are typically referred to as XS-Leaks<sup>544</sup>. Given that many of these attacks were only discovered in the last few years, the mechanisms used to tackle them obviously are very recent as well, which might explain the relatively low adoption rate. Nevertheless, compared to last year<sup>545</sup>, the cross-origin policies have significantly increased in adoption.

The `Cross-Origin-Resource-Policy`, which is used to indicate to the browser how a resource should be included (cross-origin, same-site or same-origin), is now present on 106,443 (1.5%) sites, up from 1,712 sites last year<sup>546</sup>. The most likely explanation for this is that cross-origin isolation<sup>547</sup> is a requirement for using features such as `SharedArrayBuffer` and high-resolution timers and that requires setting the site's `Cross-Origin-Embedder-Policy` to

542. <https://almanac.httparchive.org/en/2020/security#preventing-xss-attacks-through-csp>

543. [https://en.wikipedia.org/wiki/Spectre\\_\(security\\_vulnerability\)](https://en.wikipedia.org/wiki/Spectre_(security_vulnerability))

544. <https://xsleaks.dev>

545. <https://almanac.httparchive.org/en/2020/security#defending-against-xs-leaks-with-cross-origin-policies>

546. <https://almanac.httparchive.org/en/2020/security#defending-against-xs-leaks-with-cross-origin-policies>

547. <https://web.dev/cross-origin-isolation-guide/>

`require-corp`. In essence, this requires all loaded subresources to set the `Cross-Origin-Resource-Policy` response header for those sites wishing to use those features.

Consequently, several<sup>548</sup> CDNs<sup>549</sup> now set the header with a value of `cross-origin` (as CDN resources are typically meant to be included in a cross-site context). We can see that this is indeed the case, as 96.8% of sites set the CORP header value to `cross-origin`, compared to 2.9% that set it to `same-site` and 0.3% that use the more restrictive `same-origin`.

With this change, it is no surprise that the adoption of `Cross-Origin-Embedder-Policy` is also steadily increasing: in 2021, 911 sites enabled this header—significantly more than the 6 sites of last year. It will be interesting to see how this will further develop next year!

Finally, another anti-XS-Leak header, `Cross-Origin-Opener-Policy`, has also seen a significant boost compared to last year. We found 15,727 sites that now enable this security mechanism, which is a significant increase compared to last year when only 31 sites were protected from certain XS-Leak attacks.

## Web Cryptography API

Security has become one of the central issues in web development. The Web Cryptography API<sup>550</sup> W3C recommendation was introduced in 2017 to perform basic cryptographic operations (e.g., hashing, signature generation and verification, and encryption and decryption) on the client-side, without any third-party library. We analyzed the usage of this JavaScript API.

548. <https://github.com/cdnjs/cdnjs/issues/13782>  
549. <https://github.com/jsdelivr/bootstrapcdn/issues/1495>  
550. <https://www.w3.org/TR/WebCryptoAPI/>

Cryptography API	Desktop	Mobile
<i>CryptoGetRandomValues</i>	70.4%	67.4%
<i>SubtleCryptoDigest</i>	0.4%	0.5%
<i>SubtleCryptoEncrypt</i>	0.4%	0.3%
<i>CryptoAlgorithmSha256</i>	0.3%	0.3%
<i>SubtleCryptoGenerateKey</i>	0.3%	0.2%
<i>CryptoAlgorithmAesGcm</i>	0.2%	0.2%
<i>SubtleCryptoImportKey</i>	0.2%	0.2%
<i>CryptoAlgorithmAesCtr</i>	0.1%	< 0.1%
<i>CryptoAlgorithmSha1</i>	0.1%	0.1%
<i>CryptoAlgorithmSha384</i>	0.1%	0.2%

Figure 12.26. Top used cryptography APIs.

The popularity of the functions remains almost the same as the previous year: we record only a slight increase of 0.7% (from 71.8% to 72.5%). Again, this year `Crypto.getRandomValues` is the most popular cryptography API. It allows developers to generate strong pseudo-random numbers. We still believe that Google Analytics has a major effect on its popularity since the Google Analytics script utilizes this function.

It should be noted that since we perform passive crawling, our results in this section will be limited by not being able to identify cases where any interaction is required before the functions are executed.

## Utilizing bot protection services

Many cyberattacks are based on automated bot attacks and interest in it seems to have increased. According to the Bad Bot Report 2021<sup>551</sup> by Imperva, the number of bad bots has increased this year by 25.6%. Note that the increase from 2019 to 2020 was 24.1%—according to the previous report<sup>552</sup>. In the following table, we present our results on using measures by websites to protect themselves from malicious bots.

551. <https://www.imperva.com/blog/bad-bot-report-2021-the-pandemic-of-the-internet/>

552. <https://www.imperva.com/blog/bad-bot-report-2020-bad-bots-strike-back/>

<b>Service provider</b>	<b>Desktop</b>	<b>Mobile</b>
<i>reCAPTCHA</i>	10.2%	9.4%
<i>Imperva</i>	0.3%	0.3%
<i>Sift</i>	0.1%	0.1%
<i>Signifyd</i>	0.03%	0.03%
<i>hCaptcha</i>	0.03%	0.02%
<i>Forter</i>	0.03%	0.03%
<i>TruValidate</i>	0.03%	0.02%
<i>Akamai Web Application Protector</i>	0.02%	0.02%
<i>Kount</i>	0.02%	0.02%
<i>Konduto</i>	0.02%	0.02%
<i>PerimeterX</i>	0.02%	0.01%
<i>Tencent Waterproof Wall</i>	0.01%	0.01%
<i>Others</i>	0.03%	0.04%

Figure 12.27. Usage of bot protection services by provider.

Our analysis shows that under 10.7% of desktop websites, and 9.9% of mobile websites use a mechanism to fight malicious bots. Last year those numbers were 8.3% and 7.3%, so this is approximately a 30% increase compared to the previous year. This year, too, we identified more bot protection mechanisms for desktop versions than mobile versions (10.8% vs. 9.9%)

We also see new popular players as bot protection providers in our dataset (e.g., hCaptcha).

## Drivers of security mechanism adoption

There are many different influences that might cause a website to invest more in their security posture. Examples of such factors are societal (e.g., more security-oriented education in certain countries, or laws that take more punitive measures in case of a data breach), technological (e.g., it might be easier to adopt security features in certain technology stacks, or certain vendors might enable security features by default), or threat-based (e.g., widely popular websites may face more targeted attacks than a website that is little known). In this section, we

try to assess to what extent these factors influence the adoption of security features.

## Where website's visitors connect from



Figure 12.28. Adoption of HTTPS per country.

Although we can see that the adoption of HTTPS-by-default is generally increasing, there is still a discrepancy in adoption rate between sites depending on the country most of the visitors originate from.

We find that compared to last year<sup>553</sup>, the Netherlands has now made it into the top 5, which means that the Dutch are relatively more protected against transport layer attacks: 95.1% of

553. <https://almanac.httparchive.org/en/2020/security#country-of-a-websites-visitors>

the sites frequently visited by people in the Netherlands has HTTPS enabled (compared to 93.0% last year). In fact, not only the Netherlands improved in the adoption of HTTPS; we find that virtually every country improved in that regard.

It is also very encouraging to see that several of the countries that performed worst last year, made a big leap. For instance, 13.4% more sites visited by people from Iran (the strongest riser with regards to HTTPS adoption) are now HTTPS-enabled compared to last year (from 74.3% to 84.3%). Although the gap between the best-performing and least-performing countries is becoming smaller, there are still significant efforts to be made.



Figure 12.29. Adoption of CSP and XFO per country.

When looking at the adoption of certain security features such as CSP and X-Frame-Options, we can see an even more pronounced difference between the different countries, where the sites from top-scoring countries are 2-4 times more likely to adopt these security

features compared to the least-performing countries. We also find that countries that perform well on HTTPS adoption tend to also perform well on the adoption of other security mechanisms. This is indicative that security is often thought of holistically, where all different angles need to be covered. And rightfully so: an attacker just needs to find a single exploitable vulnerability whereas developers need to ensure that every aspect is tightly protected.

## Technology stack

<b>Technology</b>	<b>Security features enabled by default</b>
Automattic (PaaS)	Strict-Transport-Security (97.8%)
Blogger (Blogs)	X-Content-Type-Options (99.6%), X-XSS-Protection (99.6%)
Cloudflare (CDN)	Expect-CT (93.1%), Report-To (84.1%)
Drupal (CMS)	X-Content-Type-Options (77.9%), X-Frame-Options (83.1%)
Magento (E-commerce)	X-Frame-Options (85.4%)
Shopify (E-commerce)	Content-Security-Policy (96.4%), Expect-CT (95.5%), Report-To (95.5%), Strict-Transport-Security (98.2%), X-Content-Type-Options (98.3%), X-Frame-Options (95.2%), X-XSS-Protection (98.2%)
Squarespace (CMS)	Strict-Transport-Security (87.9%), X-Content-Type-Options (98.7%)
Sucuri (CDN)	Content-Security-Policy (84.0%), X-Content-Type-Options (88.8%), X-Frame-Options (88.8%), X-XSS-Protection (88.7%)
Wix (Blogs)	Strict-Transport-Security (98.8%), X-Content-Type-Options (99.4%)

Figure 12.30. Security features adoption by various technology.

Another factor that can strongly influence the adoption of certain security mechanisms is the technology stack that's being used to build a website. In some cases, security features may be enabled by default, or for some blogging systems the control over the response headers may be out of the hands of the website owner and a platform-wide security setting may be in place.

Alternatively, CDNs may add additional security features, especially when these concern the transport security. In the above table, we've listed the nine technologies that are used by at least 25,000 sites, and that have a significantly higher adoption rate of specific security mechanisms. For instance, we can see that sites that are built with the Shopify e-commerce system have a very high (over 95%) adoption rate for seven security-relevant headers:

`Content-Security-Policy`, `Expect-CT`, `Report-To`, `Strict-Transport-Security`, `X-Content-Type-Options`, `X-Frame-Options`, and `X-XSS-Protection`.

# 7

Figure 12.31. The number of security features with over 95% adoption rate on Shopify sites.

It is great to see that despite the variability in these content that use these technologies, it is still possible to uniformly adopt these security mechanisms.

# 83.1%

Figure 12.32. The percentage of Drupal sites that keep the default XFO header.

Another interesting entry in this list is Drupal, whose websites have an adoption rate of 83.1% for the `X-Frame-Options` header (a slight improvement compared to last year's 81.8%). As this header is enabled by default<sup>554</sup>, it is clear that the majority of Drupal sites stick with it, protecting them from clickjacking attacks. Note that, while it makes sense to keep the `X-Frame-Options` header for compatibility with older browsers in the near term, site owners should consider transitioning to the recommended `Content-Security-Policy` header directive `frame ancestors` for the same functionality.

An important aspect to explore in the context of the adoption of security features, is the diversity. For instance, as Cloudflare is the largest CDN provider, powering millions of websites (see the CDN chapter for further analysis on this). Any feature that Cloudflare enables by default will result in a large overall adoption rate. In fact, 98.2% of the sites that employ the `Expect-CT` feature are powered by Cloudflare, indicating a fairly limited distribution in the adoption of this mechanism.

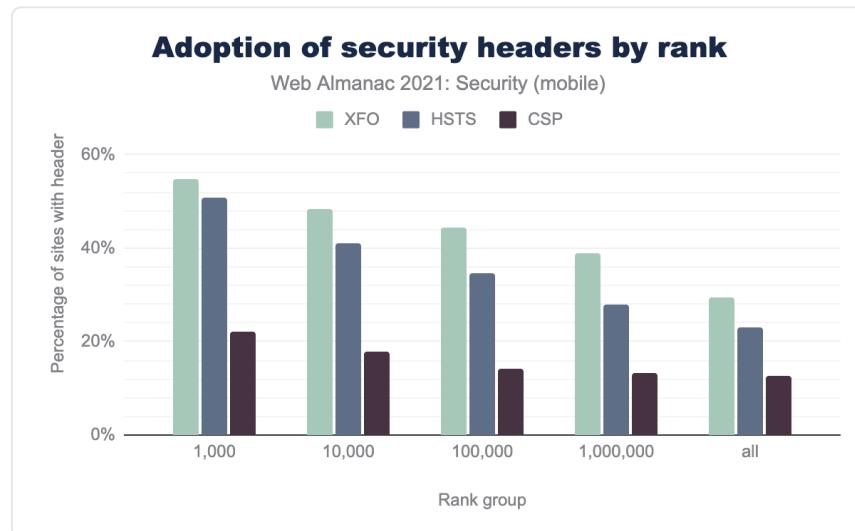
However, overall, we find that this phenomenon of a single actor like a Drupal or Cloudflare being a top technological driver of a security feature's adoption is an outlier and appears less

<sup>554</sup>. <https://www.drupal.org/node/2735873>

common over time. This means that an increasingly diverse set of websites is adopting security mechanisms, and that more and more web developers are becoming aware of their benefits. For example, last year 44.3% of the sites that set a Content Security Policy were powered by Shopify, whereas this year, Shopify is only responsible for 32.9% of all sites that enable CSP. Combined with the generally growing adoption rate, this is great news!

## Website popularity

Websites that have many visitors may be more prone to targeted attacks given that there are more users with potentially sensitive data to attract attackers. Therefore, it can be expected that widely visited websites invest more in security in order to safeguard their users. To evaluate whether this hypothesis is valid, we used the ranking provided by the Chrome User Experience Report, which uses real-world user data to determine which websites are visited the most (ranked by top 1k, 10k, 100k, 1M and all sites in our dataset).



*Figure 12.33. Prevalence of security headers set in a first-party context by rank.*

We can see that the adoption of certain security features, X-Frame-Options (XFO), Content Security Policy (CSP), and Strict Transport Security (HSTS), is highly related to the ranking of sites. For instance, the 1,000 top visited sites are almost twice as likely to adopt a certain security header compared to the overall adoption. We can also see that the adoption rate for each feature is higher for higher-ranked websites.

We can draw two conclusions from this: on the one hand, having better “security hygiene” on sites that attract more visitors benefits a larger fraction of users (who might be more inclined to

share their personal data with well-known trusted sites). On the other hand, the lower adoption rate of security features on less-visited sites could be indicative that it still requires a substantial investment to (correctly) implement these features. This investment may not always be feasible for smaller websites. Hopefully, we will see a further increase in security features that are enabled by default in certain technology stacks, which could further enhance the security of many sites without requiring too much effort from web developers.

## Malpractices on the web

Cryptocurrencies have become an increasingly familiar part of our modern community. Global cryptocurrency adoption has been skyrocketing<sup>555</sup> since the beginning of the pandemic. Due to its economic efficiency, cybercriminals have also become more interested in cryptocurrencies. That has led to the creation of a new attack vector: cryptojacking<sup>556</sup>. Attackers have discovered the power of WebAssembly and exploited it to mine cryptocurrencies while website visitors surf on a website.

We now show our findings in the following figure regarding cryptominer usage on the web.

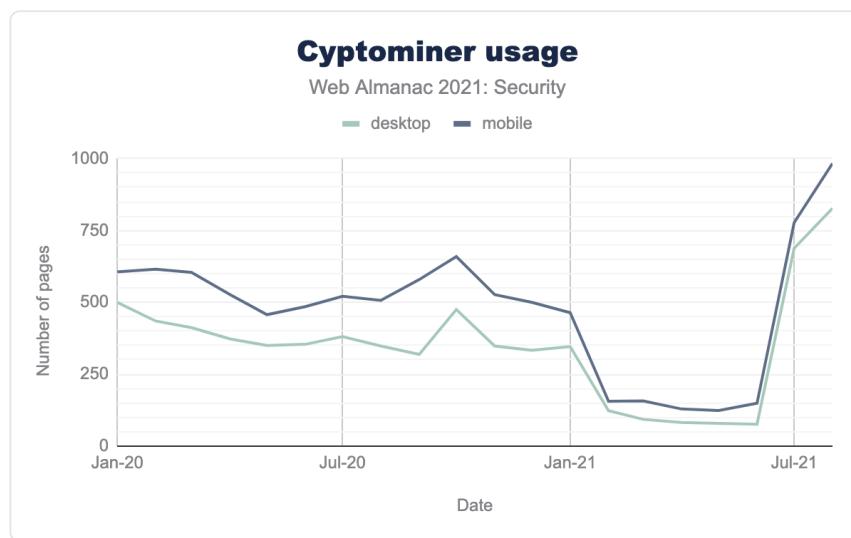


Figure 12.34. Cryptominer usage.

According to our dataset, until recently, we found a very stable decrease in the number of websites with Cryptominer. However, we are now seeing that the number of such websites has

555. <https://blog.chainalysis.com/reports/2021-global-crypto-adoption-index>

556. <https://en.wikipedia.org/wiki/Cryptojacking>

increased more than tenfold in the past two months. Such picks are very typical, for example, when widespread cryptojacking attacks take place or when a popular JS library has been infected.

We now turn to cryptominer market share in the following figure.

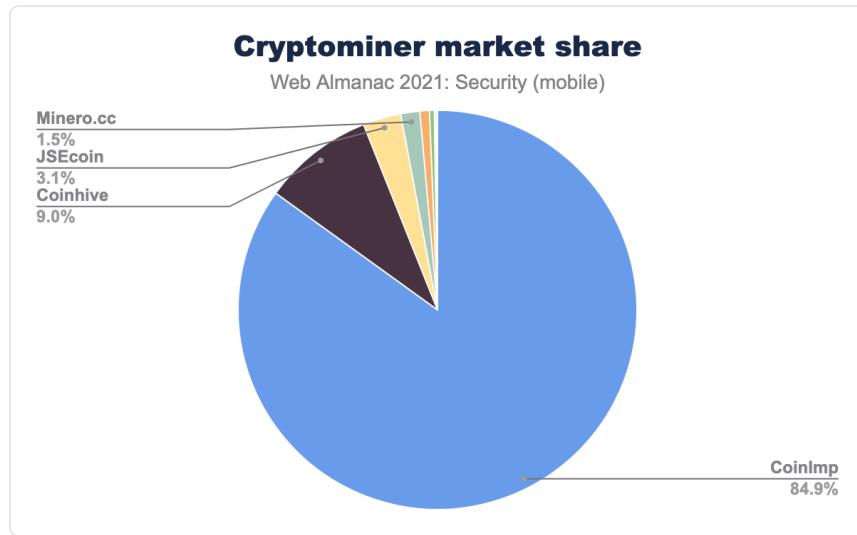


Figure 12.35. Cryptominer market share (mobile).

We see that Coinhive<sup>557</sup> has been surpassed by CoinImp as the dominant cryptomining service. One of the main reasons for this was that Coinhive was shutdown in March 2019<sup>558</sup>. Interestingly, the domain is now owned by Troy Hunt<sup>559</sup> who is now displaying aggressive banners on the website in an effort to make those sites still hosting the Coinhive script (Desktop: 5.7%, mobile: 9.0%) aware that they are—often without their knowledge. This reflects both the prevalence of Coinhive scripts even over two years after ceasing to operate, and the risks of hosting third-party resources that can be taken over should that third party cease to operate. With Coinhive's demise, CoinImp has clearly become the market leader (84.9% share).

Our results suggest that cryptojacking is still a serious attack vector, and necessary measures should be used for it.

Note that not all of these websites are infected. Website operators may also deploy this technique (instead of showing ads) to finance their website. But the use of this technique is also

557. [https://en.wikipedia.org/wiki/Monero#Mining\\_malware](https://en.wikipedia.org/wiki/Monero#Mining_malware)

558. <https://www.zdnet.com/article/coinhive-cryptojacking-service-to-shut-down-in-march-2019/>

559. <https://www.troyhunt.com/i-now-own-the-coinhive-domain-heres-how-im-fighting-cryptojacking-and-doing-good-things-with-content-security-policies/>

heavily discussed technically, legally, and ethically.

Please also note that our results may not show the actual state of the websites infected with cryptojacking. Since we run our crawler once a month, not all websites that run cryptominer can be discovered. This is the case, for example, if a website remains infected for only X days and not on the day our crawler ran.

## security.txt

`security.txt` is a file format for websites to provide a standard for vulnerability reporting. Website providers can provide contact details, PGP key, policy, and other information in this file. White hat hackers can then use this information to conduct security analyses on these websites or report a vulnerability.

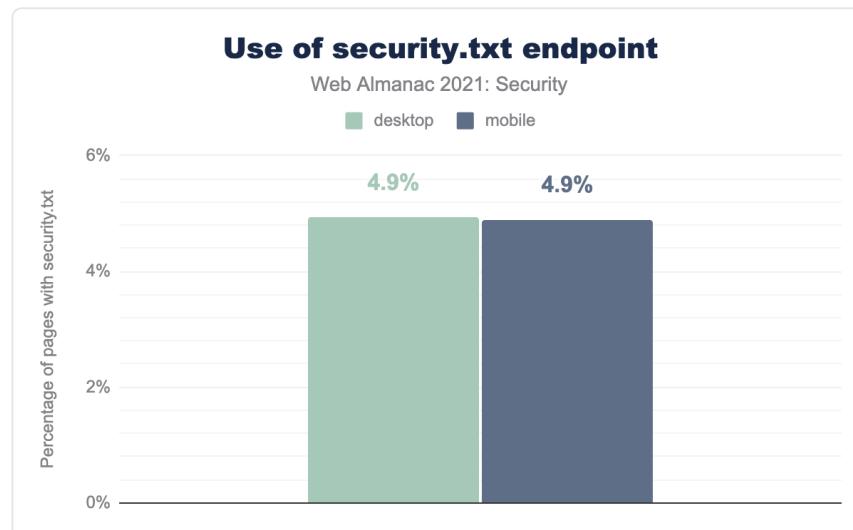


Figure 12.36. Use of `security.txt`.

We see that just under 5% of the websites return a response when asking for the `/well-known/security.txt` URL. However investigating many of these show they are basically 404 pages that are incorrectly returning a 200 status code so usage is likely much lower.



Figure 12.37. Use of security.txt properties.

We see that `Policy` is the most used property in the `security.txt` files, but even then it's only used in 6.4% of sites with a `security.txt` URL. This property includes a link to the vulnerability disclosure policy for the website that helps researchers understand the reporting practices they need to follow. This is therefore likely a better indicator of the real usage of `security.txt` since most file are expected to have a `Policy` value, meaning likely closer to 0.3% of all sites have a "real" `security.txt` file, rather than the 5% measured above.

Another interesting point is that when we look at just this subset of "real" `security.txt` URLs, Tumblr makes up 63%-65% of the usage. It looks like this is set by default for these domains to the Tumblr contact details. This is great on one hand to show how a single platform can drive adoption of these new security features, but on the other hand indicates a further reduction in actual site usage.

The other most used properties include `Canonical` and `Encryption`. `Canonical` is used to indicate where the `security.txt` file is located. If the URI used to retrieve the `security.txt` file doesn't match the list URLs in the `Canonical` fields, then the contents of the file should not be trusted. `Encryption` provides the security researchers with an encryption key that they can use for encrypted communication.

## Conclusion

Our analysis shows that the situation of web security concerning the provider side is improving

compared to previous years. For example, we see that the use of HTTPS has increased by almost 10% in the last 12 months. We also find an increase in the protection of cookies and the use of security headers.

These increases indicate we are moving safer web environment, but they do not mean our web is secure enough today. We still have to improve our situation. For example, we believe that the web community should value security headers more. These are very effective extensions to protect web environments and web users from possible attacks.

The bot protection mechanisms can also be adopted more to protect the platforms from malicious bots. Furthermore, our analysis from last year<sup>560</sup> and another study using the HTTP Archive dataset about the update behavior of websites<sup>561</sup> showed that the website components are not diligently maintained, which increases the attack surface on web environments.

We should not forget that attackers are also working diligently to develop new techniques to bypass the security mechanisms we adopt.

With our analysis, we have tried to crystallize an overview of the security of our web. As extensive as our investigation is, our methodology only allows us to see a subset of all aspects of modern web security. For example, we do not know what additional measures a site may employ to mitigate or prevent attacks such as *Cross-Site-Request-Forgery* (CSRF) or certain types of *Cross-Site-Scripting* (XSS). As such, the picture portrayed in this chapter is incomplete yet a solid directional signal of the status of web security today.

The takeaway from our analysis is that we, the web community, must continue to invest more interest and resources in making our web environments much safer—in the hope of better and safer tomorrow for all.

## Authors



### Saptak Sengupta

@Saptak013   saptaks   <https://saptaks.website/>

Saptak S is a human rights centered web developer, focusing on usability, security, privacy and accessibility topics in web development. He is a contributor and maintainer of various different open source projects like The A11Y Project<sup>562</sup>, OnionShare<sup>563</sup> and Wagtail<sup>564</sup>. You can find him blogging at [saptaks.blog](https://saptaks.blog)<sup>565</sup>.

560. <https://almanac.httparchive.org/en/2020/security#software-update-practices>

561. [https://www.researchgate.net/publication/349027860\\_Our\\_inSecure\\_Web\\_Understanding\\_Update\\_Behavior\\_of\\_Websites\\_and\\_Its\\_Impact\\_on\\_Security](https://www.researchgate.net/publication/349027860_Our_inSecure_Web_Understanding_Update_Behavior_of_Websites_and_Its_Impact_on_Security)

562. <https://www.a11yproject.com>

563. <https://onionshare.org/>

564. <https://wagtail.io/>



## Tom Van Goethem

🐦 @tomvangoethem 💬 tomvangoethem

Tom Van Goethem is a researcher at the DistriNet group<sup>566</sup> 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.

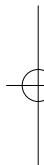
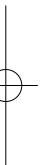


## Nurullah Demir

🐦 @nrllah 💬 nrllh 🌐 <https://internet-sicherheit.de>

Nurullah Demir is a security researcher and PhD Student at Institute for Internet Security<sup>567</sup>. His research focuses on robust web security mechanisms and adversarial machine learning.

565. <https://saptaks.blog>  
566. <https://distrinet.cs.kuleuven.be/>  
567. <https://www.internet-sicherheit.de/en/>



## Part II Chapter 13

# Mobile Web



*Written by Jamie Indigo, Dave Smart, and Ashley Berman Hale*

*Reviewed by David Fox and Fili Wiese*

*Analyzed by Ruth Everett and David Fox*

*Edited by Shaina Hantsis*

## Introduction

In January 2021, 59.5% of the global population was on the internet. Of the global 4.66 billion active internet users, 92.6% accessed the internet on a mobile device<sup>568</sup>.

With the ubiquity of mobile web tucked in our pockets, Statista<sup>569</sup> reports that 80.8% of the global population owns a smartphone. This is a relatively minor growth of 0.0% year over year. In comparison, 49.4% of the population in 2016 owned a smartphone.

In this chapter, we looked at recent trends on the mobile web including worldwide connectivity, technology adoption, and mobile-friendly feature usage.

568. <https://www.statista.com/statistics/617136/digital-population-worldwide/>  
569. <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>

## A note on methodology

When considering the challenge of how to categorize tablet experiences in relation to the mobile web, we decided to omit the data set from our analysis. Often, tablet data will be grouped into desktop or mobile. There is no uniform standard as to which it should default.

## A note on our data sources

We've used a few different data sources in this chapter:

- CrUX
- HTTP Archive
- Lighthouse
- Wappalyzer
- Akamai<sup>570</sup>

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. Learn more in our Methodology page.

## Worldwide connectivity

2021 is another year affected by the global COVID-19 pandemic, which has both affected different regions of the world differently, and the measures to combat the pandemic have varied from area to area too. Has this changed how people use their mobile devices versus laptops and computers?

## Cost of mobile web access

The financial cost of mobile web access varied greatly in 2021. One analysis<sup>571</sup> showed that the average price of 1 GB is only \$0.05 USD in Israel. The same data cost usage in Equatorial Guinea would cost a user \$49.67 USD.

Data from the Performance chapter shows the median site now weighs 2,205 KB. Using market data, What Does My Site Cost<sup>572</sup> calculated the best-case scenario price to load the median site.

---

570. <https://twitter.com/paulcalvano/status/1454866401781587969>

571. <https://www.cable.co.uk/mobiles/worldwide-data-pricing/>

572. <https://whatdoessmysitecost.com/#usdCost>

The most expensive paid loads cost Canadian users \$0.26 USD, followed by Brazil at \$0.18 USD. The same page loaded on a commonly available data plan in Poland or Russia would barely register on a users' bill, costing less than \$0.01 USD.

## Traffic to a site from mobile versus desktop (CrUX)

What percentage of traffic comes from mobile devices vs. desktop? Predicting this for any individual site can be hard, and the type of site and the industry it is in can vastly change the make-up of these different users.

### Traffic use by popularity

**77.4%**

*Figure 13.1. Percent of the 817,4923 origins in the July 2021 data received more mobile traffic than desktop traffic.*

New this year, the CrUX dataset allows us to query the most popular sites ranked by magnitude<sup>573</sup>, by traffic recorded to these origins.

<sup>573</sup>. <https://developers.google.com/web/updates/2021/03/crux-rank-magnitude>

## Percentage of Sites with more Mobile than Desktop Traffic

Web Almanac 2021: Mobile Web

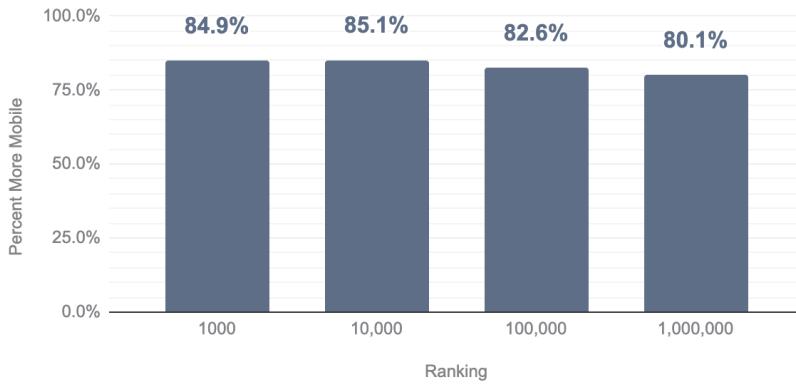


Figure 13.2. Percentage of Sites with more mobile than desktop traffic.

When grouped by CrUX ranking (the top 1,000, 10,000 and so on origins by traffic in the dataset), the more traffic a site receives, there is a slight increase of the percentage of traffic it gets from mobile, all except the top 1,000, which get slightly less (84.9% vs. 85.1%) mobile vs. desktop.

## Traffic distribution

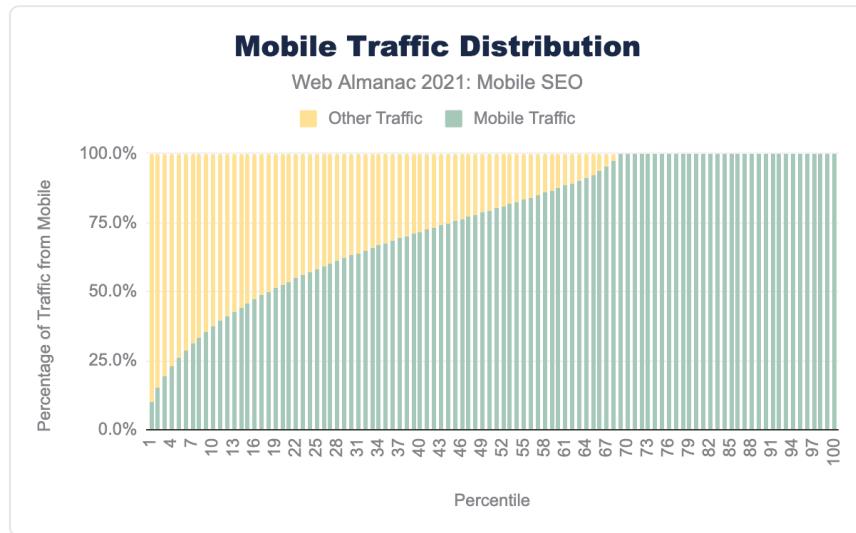


Figure 13.3. Distribution of mobile vs other traffic.

The distribution shows a similar, mobile heavy trend. At the 50th percentile, 79.4% of traffic comes from mobile devices, an increase over 77.6% in 2020, and catching up with the 79.9% percentage in 2019.

## Beyond CrUX data

A limitation of the CrUX dataset is that it can only collect data from Chrome users, who are signed in, have syncing enabled and have not disabled the *Make searches and browsing better / Sends URLs of pages you visit to Google* setting. This means that:

- Other major browsers, like Firefox and Safari are missing
- There is no data from iOS users at all (Chrome uses WebKit on iOS, like all other browsers on iOS devices)

Fortunately, there are a few other sources. Paul Calvano ran some analysis on the Akamai mPulse<sup>574</sup> real user monitoring data for July 2021. It found a slightly more even match between Mobile and Desktop traffic, at 59.4% being from mobile devices. The mPulse data is aggregated hourly, so it reveals some interesting trends

574. <https://www.akamai.com/products/mpulse-real-user-monitoring>

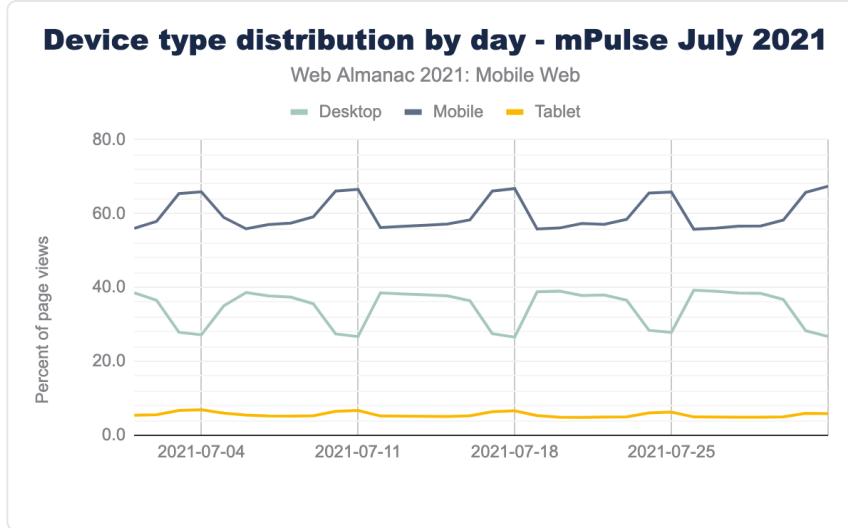
**Not all days are equal**

Figure 13.4. Device type distribution by day - mPulse July 2021.

Weekend days show a greater proportion of mobile traffic, climbing somewhere around 10% from around 55 - 56% to 65 - 67%. Globally, not every country has Monday to Friday work weeks - Sunday to Thursday is also another common pattern<sup>575</sup>, something that can be seen with a slight ramp up on Fridays, leading to a bigger jump in mobile usage on Saturdays and Sundays.

**Not all times are equal**

On weekdays, mobile usage decreases, and desktop usage increases as an overall percentage of traffic. This indicates that internet users are switching between mobile and desktop devices. Around 5 AM UTC and starts climbing again at 7 PM UTC (with a small bump around 10 / 11 AM). This aligns with working hours.

575. [https://en.wikipedia.org/wiki/Workweek\\_and\\_weekend](https://en.wikipedia.org/wiki/Workweek_and_weekend)

## Device type distribution by hour on weekdays - mPulse July 2021

Web Almanac 2021: Mobile Web



Figure 13.5. Device type distribution by hour on weekend - mPulse July 2021.

On weekends the split between mobile and desktop traffic remains more stable.

## Device type distribution by hour on weekend - mPulse July 2021

Web Almanac 2021: Mobile Web

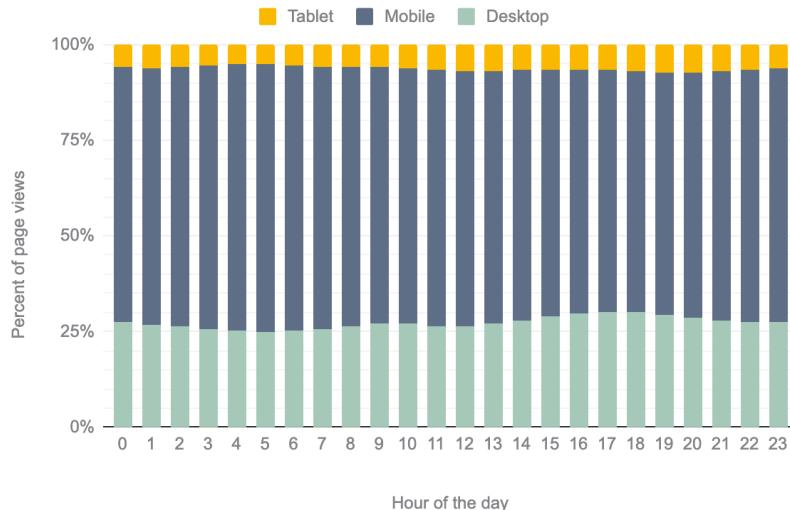


Figure 13.6. Device type distribution by hour on weekend - mPulse July 2021.

This all suggests that people who have the choice between different devices are more likely to use mobile ones in their personal time.

Cloudflare also released a great study. Like the Akamai data, this study shows a much closer split between mobile and desktop devices than the CrUX dataset. In the 30 days leading up to October 4th, 52% of traffic was mobile.

We looked for, in the past month, the country with the highest proportion of mobile Internet traffic. And the answer is... Sudan, with 83% of Internet traffic is done using mobile devices – actually it's a tie with Yemen.

– João Tomé, *Where is mobile traffic the most and least popular?*<sup>576</sup>

Cloudflare's Radar<sup>577</sup> trend reports allow them to segment traffic by geographic region, and it's interesting to see the variations regionally between the split of mobile vs. desktop, from Sudan

576. <https://blog.cloudflare.com/where-mobile-traffic-more-and-less-popular/>

577. <https://radar.cloudflare.com/>

and Yemen tying at 83% usage, compared to the Seychelles at just 29% mobile.

## Drawing conclusions

Mobile device usage remains strong, and it's apparent that despite a global trend of people being at home more than ever before (due to restrictions and advice from health authorities and governments), mobile devices remain the most popular way to access websites. The popularity of mobile over desktop seems to have regained most of the ground lost last year—itself a fairly small regression.

Naturally the figures cannot tell us the reasons behind that, but it's worth remembering that for a large amount of web users, mobile devices may be the only device available to them, and there is no choice between using a mobile or a desktop.

Whilst it can be hard to predict if your mobile traffic percentage is expected, if it seems low vs. your region and sector, it could be an indication you are under-serving this portion of your user base.

## Mobile methodology & tech stacks

While mobile web is highly used, these experiences typically have less processing power and slower internet interconnectivity. Many technologies have emerged to mitigate these limitations. These include Client Hints and APIs that identify the connection type and serve assets best suited for the connection.

In this section we will also look at overall app usage for the mobile web and how the programming languages, content management systems, and web servers compare to desktop experiences.

### Client Hints

*Client Hints* are a collection of HTTP request header fields a server can request from the client accessing it to get information on the device, its capabilities, the network conditions and other agent settings and preferences.

This gives the ability to make decisions and serve code, content and experience that's more tailored to that device.

For the mobile web, poor network conditions and lower powered devices are much more common, and sites that are proactively requesting this information are likely to be thinking

beyond merely squeezing down their desktop pages to fit on a mobile screen.

HTTP Client Hints are a relatively new, and somewhat experimental feature, with the RFC only published in February this year<sup>578</sup>. It's therefore fairly encouraging that we found 1.4% of sites are requesting at least one of these Client Hints from mobile users, compared with just 1.0% for desktop users.

Whilst we are not able to tell what the sites might do with that information, and exactly how they use these hints to tailor the experience to mobile users, asking is a good first sign.

These hints can be roughly assigned into three groups:

- **Device Client Hints:** Details of the capabilities and features of the device accessing the site.
- **Network Client Hints:** Details of the network connection between the device and the server.
- **User-Agent Hints:** Details about the agent accessing the site.

## Device Client Hints

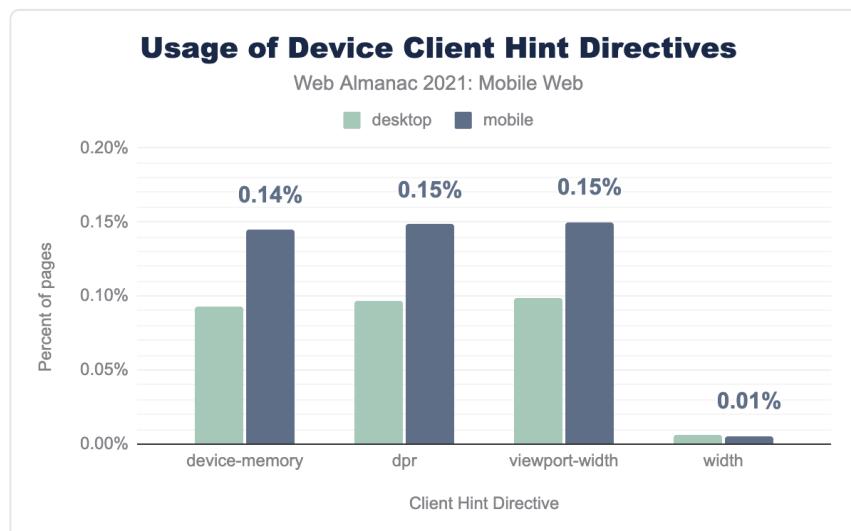


Figure 13.7. Usage of Device Client Hint directives.

578. <https://www.rfc-editor.org/rfc/rfc8942#section-3.1>

Uptake here is low, with `DPR` and `Viewport-Width` leading with 0.15% of mobile sites requesting this, `Device-Memory` a little behind at 0.14% and `Width` at just 0.0%, but this is now deprecated, the proposed replacement being `Sec-CH-Width`, we detected no sites requesting this.

Currently, only Chrome, (and Chromium based browsers like Microsoft's Edge), and Opera support these headers, with Safari and Firefox not yet onboard<sup>579</sup>.

## Network Client Hints

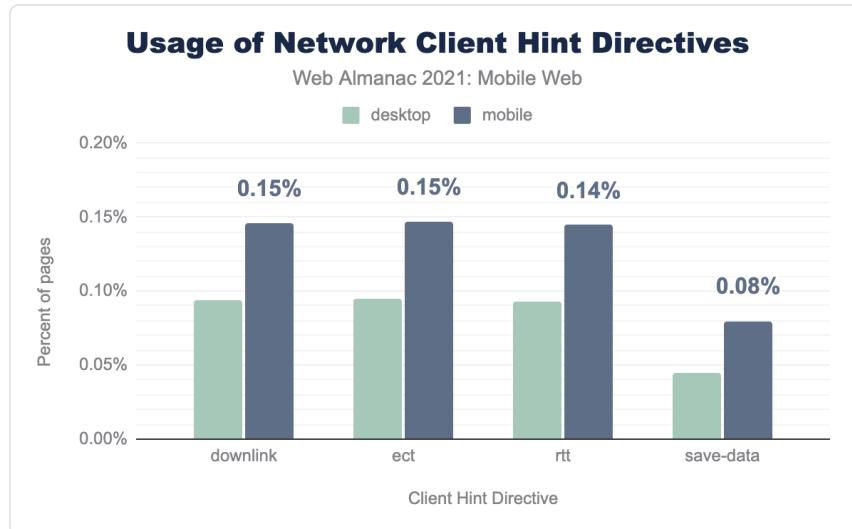


Figure 13.8. Usage of Network Client Hint directives.

Network Client Hints show a similar uptake to Device Client Hints, with `Downlink`<sup>580</sup> and `ECT`<sup>581</sup> (effective connection type) being requested by 0.2% of loads on mobile, and `RTT`<sup>582</sup> (round trip time) on 0.1% of loads on mobile.

`Save-Data` is surprisingly present less, at just 0.1% of mobile requests, seemingly a missed opportunity, given the user benefits possible, as detailed in the Google Web Fundamentals article, `Delivering Fast and Light Applications with Save-Data`<sup>583</sup>.

579. <https://caniuse.com/client-hints-dpr-width-viewport>

580. <https://developer.mozilla.org/docs/Web/HTTP/Headers/Downlink>

581. <https://developer.mozilla.org/docs/Web/HTTP/Headers/ECT>

582. <https://developer.mozilla.org/docs/Web/HTTP/Headers/RTT>

583. <https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/save-data/>

## User-Agent Client Hints

Major browsers like Chrome<sup>584</sup>, Safari<sup>585</sup> and Firefox<sup>586</sup> reducing and capping the `User-Agent` string to reduce passive fingerprinting<sup>587</sup>.

Traditionally, sites may have used this information to tailor the experience to those devices. This approach has always had some drawbacks in trying to keep up with the ever-changing landscape of devices, and the fact the user-agent string is easily changeable and spoofable.

*User-Agent Client Hints* offer a way to get this information, but unlike the Device and Network Hints do not require the server to request this via the `Accept-CH` header. This is perhaps why we detected only a tiny handful of sites requesting this.

## Network Information API and Device Memory API usage

The *Network Information API* and `Navigator.deviceMemory` offer an interface to JavaScript to gather device and connection information, similar in scope to those exposed with Client Hints.

---

584. <https://blog.chromium.org/2021/05/update-on-user-agent-string-reduction.html>

585. [https://bugs.webkit.org/show\\_bug.cgi?id=216593](https://bugs.webkit.org/show_bug.cgi?id=216593)

586. [https://bugzilla.mozilla.org/show\\_bug.cgi?id=1679929](https://bugzilla.mozilla.org/show_bug.cgi?id=1679929)

587. <https://www.w3.org/2001/tag/doc/unsanctioned-tracking-tracking-without-user-control>

## Network Information API



Figure 13.9. Usage of `NetworkInformation.effectiveType`.

We focused of mobile vs. desktop page loads making use of `NetworkInformation.effectiveType`, which returns a string based on the effective connection type, `slow-2g`, `2g`, `3g`, or `4g`. The top tier is `4g`, so could really be seen as “`4g` or faster”, including `5g` and broadband, fixed connections.

18.2% of mobile requests had page loads utilizing `NetworkInformation.effectiveType`, but surprisingly, a very slightly higher 18.4% of desktop requests detected use of this API.

## Device Memory API

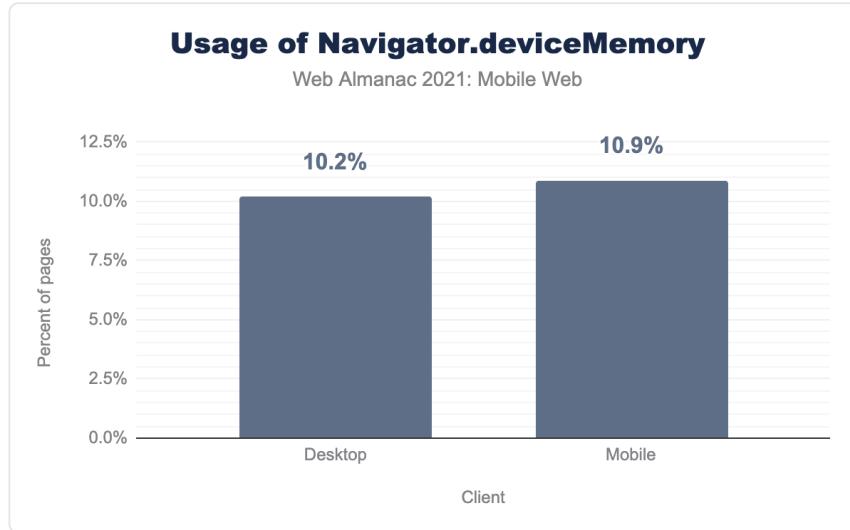


Figure 13.10. Usage of `Navigator.deviceMemory`.

This API returns an approximate amount of device memory, useful to judge what the client might be capable of handling and adapt accordingly.

10.9% of mobile page loads utilized this API, slightly higher than 10.2% for desktop loads.

Much like Client Hints, these APIs are still experimental, and also do not have universal support across browsers (source: Network Information API<sup>588</sup> & `Navigator.deviceMemory` but have much wider adoption.

One reason for wider adoption could be third-party scripts requesting these on page loads. Another reason may be ease of implementation. Setting and reading HTTP headers may be seen as more complex and more likely to involve changes to infrastructure.

## Client Hints, Network Information API and Device Memory API conclusions

For experimental APIs and features, there are already some encouraging take up of these features. Hopefully as browser support grows and the APIs move from experimental status, uptake will grow further.

588. <https://caniuse.com/netinfo>

If you have a network or device capability limited web app, and you have a significant proportion of users accessing from lower powered devices, and/or poor network connections, now might be the time to investigate if these APIs can let you offer a better user experience for them.

## App usage on the mobile web

The most commonly used libraries and technologies found on the mobile web impact performance and inform us on technology adoption.

According to Wappalyzer<sup>589</sup> data, JavaScript library JQuery is the dominant library of the mobile web, present in 84.4% of tested sites. Google is the dominant provider, holding three of the top five spots.

<b>App</b>	<b>Mobile</b>	<b>Desktop</b>	<b>Diff desktop v mobile use</b>
jQuery	84.4%	84.4%	1.0%
Google Analytics	65.4%	68.6%	3.2%
PHP	50.5%	50.5%	-0.4%
Google Font API	47.6%	47.6%	-0.1%
Google Tag Manager	43.4%	43.4%	2.6%

Figure 13.11. Popular technology usage.

Of the top five mobile web technologies, adoption rates for three were higher on desktop sites. It is reasonable to attribute lower mobile adoption rates of these apps to mobile performance initiatives as these apps are frequently flagged by Lighthouse, the open-source auditing tool recommended by Google to diagnose performance issues.

In 2021, Google added the Page Experience Ranking Signal<sup>590</sup> to its algorithm. This ranking signal is specific to search engine results pages served on mobile devices and uses aggregated data from real user page loads to measure performance.

JavaScript library JQuery is the dominant library of the mobile web, present in 84.4% of mobile page loads. Google is the dominant provider, holding three of the top five spots.

589. <https://www.wappalyzer.com/>

590. <https://developers.google.com/search/docs/advanced/experience/page-experience>

## Content Management Systems

Content management systems allow site owners to publish, update, and control content through an authenticated backend. The top five content management systems on the mobile web in 2021 were:

CMS	Mobile	Desktop
WordPress	33.6%	32.9%
Joomla	2.0%	1.7%
Drupal	1.8%	2.1%
Wix	1.6%	1.2%
Squarespace	1.0%	1.2%

Figure 13.12. Prominent mobile vs. desktop CMS.

WordPress, an open-source CMS written in PHP, was the dominant CMS in 2021. The technology appeared on 33.6% of sites.

## Comparing desktop technology adoption rates

Technology adoption rates for the mobile web moved in step with desktop. The most notable difference came in the form of third-party pixel use. 68.6% of desktop sites used Google Analytics compared to 65.4% of mobile sites.

Category	Technology	Desktop	Mobile	% higher desktop adoption rate
Analytics	Google Analytics	68.6%	65.4%	3.2%
Tag managers	Google Tag Manager	46.0%	43.4%	2.6%
Analytics	Facebook Pixel	20.6%	18.9%	1.7%
Widgets	Facebook	28.0%	26.3%	1.6%
JavaScript libraries	jQuery UI	23.8%	22.2%	1.5%

Figure 13.13. Technology with higher desktop adoption rates.

Given the changes to performance measurement and prioritization, it's reasonable to consider the absence of these JavaScript-heavy, third-party, assets as part of an intentional effort to improve mobile page experience. The Facebook Pixel analytics script was found on -1.7% fewer mobile sites than desktop.

Mobile sites were more likely to adopt certain technologies, but with a smaller margin. Blogger was found on 3.1% of mobile sites and 1.7% of desktop sites

<b>Category</b>	<b>Technology</b>	<b>Desktop</b>	<b>Mobile</b>	<b>% higher mobile adoption rate</b>
Blogs	Blogger	1.7%	3.1%	1.5%
Web servers	OpenGSE	1.7%	3.2%	1.5%
Programming languages	Python	2.2%	3.6%	1.4%
Programming languages	Java	2.8%	4.0%	1.2%

Figure 13.14. Technology with higher mobile adoption rates.

### Drawing conclusions on mobile web app usage

JavaScript via JQuery permeated the mobile web in 2021. Third-party analytics tools had a lower adoption rate on mobile.

One thing that shines through in the data is that at a CMS and web server level, mobile and desktop share a close correlation in how people develop sites, perhaps in large part to the lower overheads of responsive design, meaning one codebase for all experiences.

With WordPress not only maintaining, but extending its popularity for mobile sites, and other CMSs enjoying a similar share to the desktop experience, there's a great opportunity for CMS core improvements and optimizations to bring an outsized benefit to the whole mobile web.

This makes drives like the proposed WordPress Performance Team<sup>591</sup> important and valuable.

## Interacting with the mobile web

Attention to mobile design and friendliness are critical to reducing friction in the user journey.

591. <https://make.wordpress.org/core/2021/10/12/proposal-for-a-performance-team/>

Users navigate the mobile web with taps of their fingers rather than the more refined control provided by a mouse or trackpad.

## Alternative protocol links

The web is built on links. On the mobile web, Unique Resource Identifier<sup>592</sup> schemes beyond http/s, can allow users to complete tasks like dialing a phone number using `tel:` or starting an email with minimal friction.

The most prevalent URI schemes were `https:`, found on 93.2% of sites, and its non-secure equivalent, `http:`, appearing on 56.7%. The high use of non-secure link protocols is noteworthy as 2020 saw major announcements from browsers to protect users' safety by alerting them when content is not secure.

After web page links, the next five most used protocols in anchor href values on the mobile web are as follows:

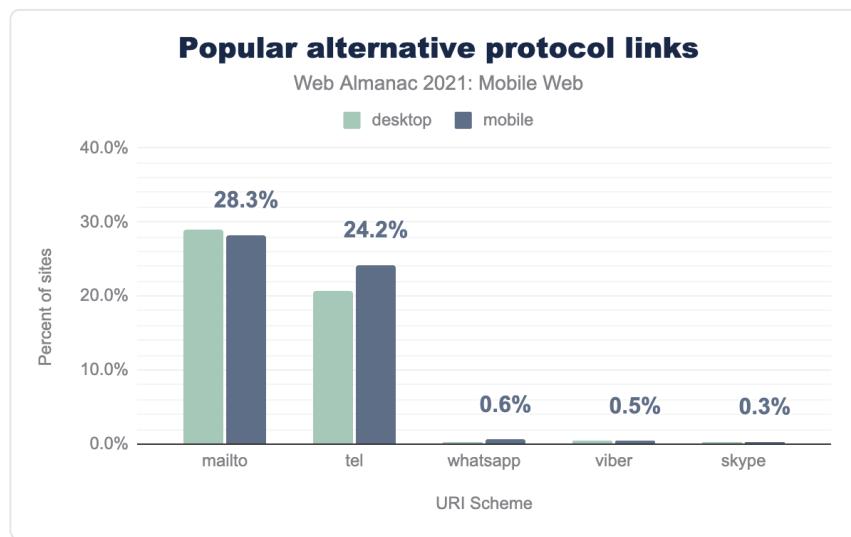


Figure 13.15. Popular alternative protocol links.

Mobile devices whilst limited in some aspects do tend to be better connected, they are a phone, have SMS and other messaging services where desktop clients may not. Usage of other link protocols past the standard `http:` / `https:` can help unlock some of these capabilities. Providing a tappable link to call or send a message without having to copy and paste makes for a

<sup>592</sup>. [https://en.wikipedia.org/wiki/Uniform\\_Resource\\_Identifier](https://en.wikipedia.org/wiki/Uniform_Resource_Identifier)

smoother, more integrated user interaction.

### mailto

`mailto:` invokes the users chosen email client, clicking:

```
<a href="mailto:enquiries@example.com?subject=Enquiring about Red  
Widgets">  
    enquiries@example.com  
</a>
```

Would prefill an email with the specified email address and subject line. Helpful on mobile, but also relevant for desktop too.

### tel

`tel:` invokes a call:

```
<a href="tel:+44123467890">  
    Call +44 (0)123 4567890  
</a>
```

Would open the phone app, ready to dial that number. This saves copy / paste and reduces friction if your business values phone leads or enquiries.

### sms

`sms:` invokes the clients default SMS messaging app:

```
<a href="sms:+441234567890">  
    Text Us  
</a>
```

When clicked would prefill a message with the right number, you can also prefill the message

body. This fell out of the top 5, with just 0.3% of mobile site loads utilizing this.

## Other messaging apps

Other messaging apps can register a protocol to have a `<a href="">` open them, as seen in the table above, WhatsApp and Viber are the two leading ones here, outstripping the native `sms:` app usage.

### Alternative protocol links conclusions

`mailto:` has a long history on the internet, right back to 1994<sup>593</sup>, but it's encouraging to see `tel:` reach 24% usage, not a long way behind, given its additional usefulness on mobile devices.

It's surprising to see `sms` with such small uptake, and disappointing that its uptake is below proprietary apps like WhatsApp and Viber.

SMS is more likely to be available as default and require no additional installations, so seemingly more accessible. However, WhatsApp and Viber messages are free, while SMS messages may incur charges from the user's mobile provider. This could explain that relative popularity.

If you aren't using some of the extended capabilities for communication that protocols past `https:` can offer your users, and it's a good fit for your mobile website, these could offer a simple, user friendly, low development benefit.

## Input fields

While URI schemes allow users to take actions from a website, input fields allow users to provide information to a website.

Input elements are one of the most powerful and complex features in HTML. Input elements are used to create interactive controls for web-based forms. Web users experience these elements such as buttons, checkboxes, calendars, search, and other elements which allow control of a page's content based on user input.

593. <https://datatracker.ietf.org/doc/html/rfc1738#section-3>

# 71.5%

Figure 13.16. Percent of mobile pages using inputs.

71.5% of mobile pages tested contained inputs. This is slightly higher than the 71.1% of desktop.

## Type declarations

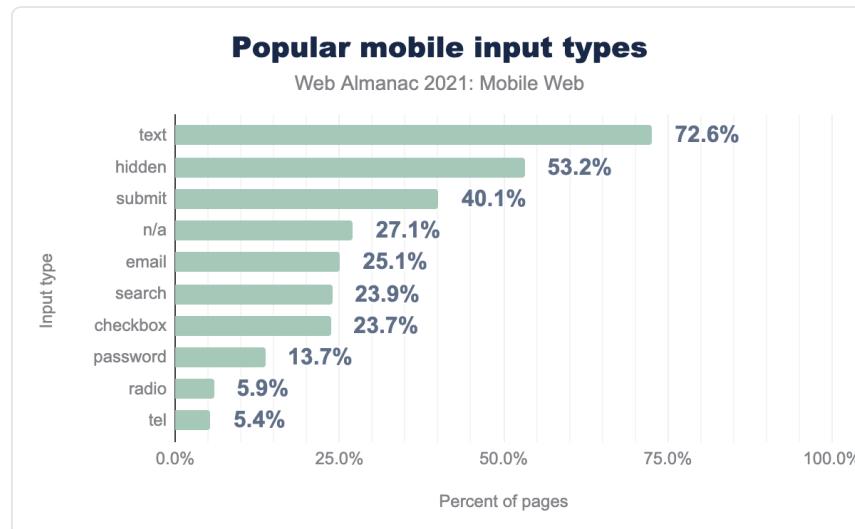


Figure 13.17. Popular mobile input types.

We can track occurrences of interactive controls created by `input` by looking for the `type` attribute. The `type` attribute is the most important because it controls how the `input` element works. The `type` attribute value was declared on 70.9% of tested sites.

If the `type` attribute is not present the `input` defaults to `text`, a single line text field. In analysis of pages using `input` elements, 27.1% of those pages did not declare an `input` type and used the default `text` string value.

Out of all pages using inputs, 72.6% contained at least one `text` input type. This was the most used.

The declared `text` value combined with the fallback value indicates that 99.7% of sites using

input elements capture a text value.

## Advanced input types

# 44.8%

Figure 13.18. Percent of mobile pages using inputs.

Of pages with at least one input, 44.8% of them use one or more “advanced input types”.

Advanced input types include `color`, `date`, `datetime-local`, `email`, `month`, `number`, `range`, `reset`, `search`, `tel`, `time`, `url`, `week`, `datalist`.

### Telephone

5.4% of pages asked users for their telephone number. For mobile users, navigating from the alpha to numeric keyboard is a high friction point. 62.6% of pages soliciting a telephone number used an input field missing the `type=tel` value.

### Email

The `email` input type requires the user to submit a valid email address. A non-email value entered in the form prompts an error to display when the form is submitted.

25.1% of pages contained at least one field asking users for their email.

Email collection is often a key micro conversion in the user journey so capturing it with minimal friction benefits the site with a higher conversion rate. Even with this clear business value, 42% of pages which ask for user emails do not use the `type=email` input type on at least one instance.

### Search input

Site search is a powerful tool in navigating users to their desired content. Search inputs are text fields functionally identical to text. The main difference between search and text input fields is how they are handled by the browser.

Use of the search input type can trigger a cross icon which allows users to quickly clear existing query text. Many modern browsers also store search queries across domains. When the search

type is denoted, stored queries can be used to autocomplete the field.

23.9% of tested pages contained a search input field. It is worth noting that these fields may be present though using a text or undeclared input type. This is a slight increase over 2020 which saw 17% of sites using search input.

Business value appears to impact input type adoption. Ecommerce sites have a vested interest in swiftly moving users to a desired product in order to meet the business goal of a transaction.

43.3% of tested ecommerce sites use search input on their mobile experience. Interestingly, this is higher than 42.6% of sites using the input type for desktop clients.

## Autocomplete

The `autocomplete` attribute allows some control over how forms and inputs work with browsers autofill features. There are a number of options, from disabling it entirely, to providing hints as to what to autofill, like a name, or street address.

Inputting text and data on mobile devices is a generally more tedious process than on a device with a full keyboard, so autofill becomes an even more useful and time saving feature than for desktop users. Google discovered<sup>594</sup> a 25% increase in form submission when autofill is used.

For mobile page loads, 24.8% of pages utilized the `autocomplete` attribute, lower than the 27% of desktop page loads.

As the HTTP Archive data captures only homepages, usage could be much higher in checkout, contact and other places that are likely to require inputs, but it is perhaps disappointing to see lower usage on mobile experiences, where arguably it is the most useful.

## Input field conclusions

Input type declarations are critical in reducing friction. If an input element is marked up using the appropriate type, input elements can prompt different keyboards to improve the experience. The boon to user experience makes the low-lift adoption of input types a meaningful investment.

The low rates of adoption for input types like telephone and email are surprising given the ubiquity of input fields on the mobile web. This gap between business goals and the user experience illustrates that user experience on the mobile web is critical. The greatest opportunities from websites may not come from in-house feature development, but rather

<sup>594</sup>. <https://www.youtube.com/watch?v=m2a9hIUFRhg&t=1433s>

leveraging the growing functionalities natively available in modern browsers.

## Accessibility on the mobile web

The pandemic forced humans around the world to isolate themselves from friends, family, and community. The number of persons facing disabilities also increased due to post-COVID conditions<sup>595</sup>. This shift forced digital spaces to the new default as in-person services, commerce, and communication were disrupted.

The goal of accessibility is to create web experiences which provide feature and information parity to all users. Users on the mobile benefit from accessibility as accessibility practices make information available to people using slow internet connections, or who have limited or expensive data plans.

### ARIA roles

Accessible Rich Internet Applications (ARIA) is a set of attributes that supplement HTML so that commonly used interactions and widgets can be passed to assistive technologies. These attributes are also useful to search engines in understanding page content<sup>596</sup>.

When a site is accessed using assistive technology, an element's ARIA role communicates information about how the user can interact.

---

595. [https://www.hhs.gov/civil-rights/for-providers/civil-rights-covid19/guidance-long-covid-disability/index.html#footnote10\\_0ac8mdc](https://www.hhs.gov/civil-rights/for-providers/civil-rights-covid19/guidance-long-covid-disability/index.html#footnote10_0ac8mdc)

596. <https://webaim.org/blog/web-accessibility-and-seo/>



Figure 13.19. Top 10 most common ARIA roles.

The most prevalent ARIA role in 2021 was `button` which appeared on 29% of sites. The `button` role indicates a clickable element that triggers a response when activated by users.

While over 71% of mobile sites have interactive-controls for web-based forms, the most commonly adopted ARIA attribute, `aria-label`, only appeared on 11.2% of tested sites. This accessibility-focused attribute is used to label input with a text string.

### Color contrast

A lack of color contrast impacts users with color blindness as well as low color sensitivity, a condition common in older people. Sufficient color contrast allows for equal access to content and a positive impact to business goals. In a case study by Google, ecommerce site Eastpak saw a 20% increase in click through rate<sup>597</sup> when call-to-action buttons used sufficient contrast between text color and its background.

<sup>597</sup>. <https://www.thinkwithgoogle.com/intl/en-154/marketing-strategies/app-and-mobile/5-lessons-eastpak-learned-its-mobile-audience/>

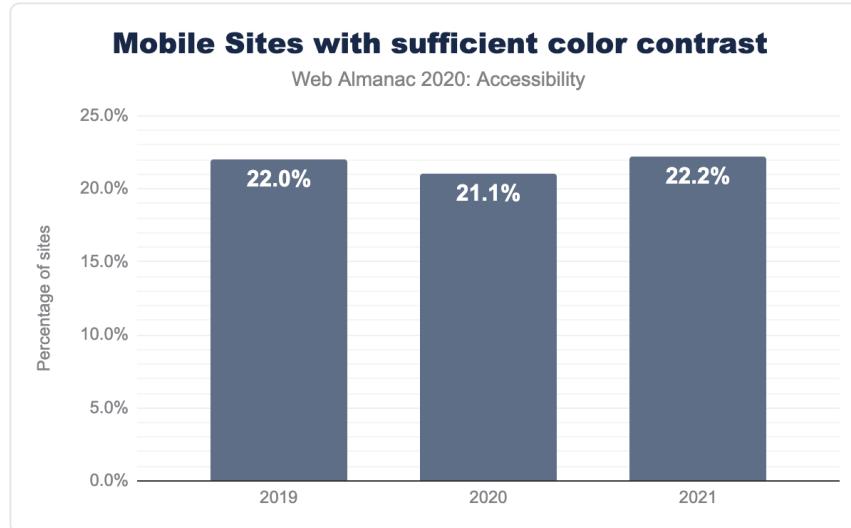


Figure 13.20. Mobile Sites with sufficient color contrast.

Despite the potential for increased conversion, 77.8% of sites failed Lighthouse audits for use of sufficient color contrast. This is a slight improvement year over year.

## Tap targets

Tap targets are elements that respond to user input. These include links, buttons, form fields, and many others.

In order for effective user interactions, tap targets need to be both appropriately sized and spaced apart from other tap targets on the page. Interactive elements should be at least 48x48 pixels and have a padding of at least 8 pixels separating them from other interactive elements.

**39.3%**

Figure 13.21. Percent of mobile sites using sufficiently-sized tap targets.

Overall, 39.3% of sites tested used sufficiently-sized mobile tap targets. Tap target adoption was consistent across domain rank groupings. This is a slight increase from 2020, which saw 36.3% of tap targets properly sized.

## Zoom and scaling

The Viewport meta element is important to inform a browser how to lay out the page on a user's device. It's also possible to configure this by adding the `user-scalable="no"` or a small `maximum-scale:` parameter to either prevent totally, or limit the ability for users to zoom in on the content. On mobile devices, this is commonly pinch zooming.

Preventing the ability to zoom in is an issue for low vision users and is something that would fail<sup>598</sup> the WCAG 2.0 guidance.

Disappointingly, 29.4% of mobile page loads fail this requirement, and contained a viewport that prevented zooming, this is a slight improvement over the 30.7% (source: 2020 Web Almanac Accessibility<sup>599</sup> chapter).

Things look even worse when looking at the usage by domain ranking.



Figure 13.22. Disabled zooming and scaling by domain rank.

The more popular sites are more likely to fail this, meaning that overall, more users are reaching mobile sites that are not compliant.

598. <https://dequeuniversity.com/rules/axe/3.3/meta-viewport>

599. <https://almanac.httparchive.org/en/2020/accessibility#zooming-and-scaling>

## Accessibility conclusions

When the web is accessible, more people can perceive, understand, navigate, interact with, and contribute to the web. Equal and inclusive access must be prioritized in order to keep pace with the growth and necessity of web access.

The areas we've covered here are a small part of accessibility. ARIA, zooming, and color contrasts are bare minimum requirements. A study from W3C's Web Accessibility Initiative<sup>600</sup> show that 15% of the world's population (over 1 billion people) have a recognized disability. Far more may go unregistered or will develop a disability at some point in their lives that may affect their ability to access your sites. Accessibility isn't for a tiny minority.

The poor adoption of good accessibility practice creates a technical barrier to these users that should disturb us as humans, aside from the clear commercial opportunity of properly catering for this sizable group of potential users.

In many jurisdictions, accessibility is not just good practice.

*Last year lawsuits related to the Americans with Disabilities Act were up 20%.<sup>601</sup>*

*– Web Almanac 2021 Accessibility Chapter*

To learn more about accessibility on the mobile web, visit the Accessibility chapter.

## Mobile Search Engine Optimization (SEO)

For any website, acquisition is a critical step, the best optimized mobile website is no different to the worse if no one finds and visits it.

The primary avenue of discovery is quite likely to be from a search engine, along with social media and links from other websites.

With search engines being the primary source of acquisition for many sites, and a still sizeable one for many more, SEO is an important consideration for pretty much every site.

There are some mobile specific areas and concerns in SEO.

---

600. <https://www.w3.org/WAI/business-case/#increase-market-reach>  
601. <https://info.usablenet.com/2020-report-on-digital-accessibility-lawsuits>

## Mobile-first index

Google recognizes that the predominant method of accessing the web is now mobile, and now index websites predominately with a mobile user-agent<sup>602</sup>. Since July 2019, all new sites have been indexed this way, and most existing sites have now transitioned to mobile-first indexing too.

This means that if you have content or markup that's only served to desktop devices, google will no longer index that part.

## Mobile-friendliness

Both Google<sup>603</sup> and Bing<sup>604</sup>, among other search engines, use some concept of mobile friendliness as a direct ranking signal. This mostly comprises testing to make sure that the content fits in the viewport, text is legible and tap targets are of a reasonable size.

Google offers a mobile-friendly test<sup>605</sup>, as does Bing<sup>606</sup> to help diagnose if your pages are passing.

The recommended way of achieving this is using responsive web design, web.dev have a great learning resource<sup>607</sup>.

## Core Web Vitals & Page Experience

On July 15th 2021, Google announced that they were rolling out the Page Experience Ranking Update<sup>608</sup>. This comprises a few different signals, including mobile-friendliness, with the major new additions being the Core Web Vitals metrics<sup>609</sup>.

Of particular interest to the mobile web is that the Core Web Vitals part is mobile specific<sup>610</sup>, these metrics only play a part in the mobile results so far, although a roll out to desktop is planned in February 2022<sup>611</sup>.

You can learn more about the role of mobile-friendliness and the Core Web Vitals in SEO over in the SEO chapter.

---

602. <https://developers.google.com/search/mobile-sites/mobile-first-indexing>  
 603. <https://developers.google.com/search/blog/2015/04/rolling-out-mobile-friendly-update>  
 604. <https://blogs.bing.com/webmaster/2015/11/12/mobile-friendly-test>  
 605. <https://search.google.com/test/mobile-friendly>  
 606. <https://www.bing.com/webmaster/tools/mobile-friendliness>  
 607. <https://web.dev/learn/design/>  
 608. <https://developers.google.com/search/blog/2021/04/more-details-page-experience>  
 609. <https://web.dev/vitals/>  
 610. <https://support.google.com/webmasters/thread/104436075/core-web-vitals-page-experience-faqs-updated-march-2021>  
 611. <https://developers.google.com/search/blog/2021/11/bringing-page-experience-to-desktop>

## Mobile performance

A mobile device is likely to be lower powered, and on a slower and less reliable network connection than desktop devices. Given these circumstances, performance can be a bigger challenge and a bigger priority.

### Loading performance

Grabbing the attention of your newly acquired user or keeping the attention of a returning user begins with making sure they see the important content of the site quickly.

#### Largest Contentful Paint

Largest Contentful Paint<sup>612</sup> (LCP) is a metric designed to capture this experience (and is one of the Core Web Vitals). It's a measure of when the largest element in the viewport is rendered, it's limited to `<img>`, `<image>` inside an `<svg>`, `<video>` (if the poster is set), a block element with a background image, or a text block.

An LCP of 2.5 seconds or less is considered a good score.



Figure 13.23. LCP performance by device. Data from the Performance chapter.

<sup>612</sup> <https://web.dev/lcp/>

The data shows that just 45% of mobile page loads recorded in the CrUX dataset are meeting the 2.5 second or under target, far lower than the 60% desktop achieves.

It does represent a small improvement from 2020, where only 43% of mobile page loads<sup>613</sup> met the 2.5 second or under threshold.

There are clearly bigger challenges to achieving good LCP scores for the mobile demographic, but one worth chasing. A recent study from Vodafone<sup>614</sup> showed that a reduction of just 8% in LCP times lead to increased conversions of 31%. Performance can have a direct effect on revenue.

## Images

Many different assets can and do affect load times on mobile, CSS & JavaScript can all play a big part. But a big factor remains images.

Too often an approach to responsive web design is to supply an image whose native size is appropriate for desktop users, and just scale it to the screen with CSS.

### Appropriately sized images

56.6%

*Figure 13.24. Percent of mobile page loads that had appropriately sized images*

This is sadly a step back from 58.8% in 2020. That's 43.4% of mobile users getting the wrong size images.

### Responsive images

Images can be served responsively<sup>615</sup> too, the `srcset` attribute, and the `<picture>` element allow appropriately sized, and appropriately formatted images to be specified, allowing the browser to download the one that best matches the screen and device.

613. <https://almanac.httparchive.org/en/2020/performance#lcp-by-device>

614. <https://web.dev/vodafone/>

615. [https://developer.mozilla.org/docs/Learn/HTML/Multimedia\\_and\\_embedding/Responsive\\_images](https://developer.mozilla.org/docs/Learn/HTML/Multimedia_and_embedding/Responsive_images)

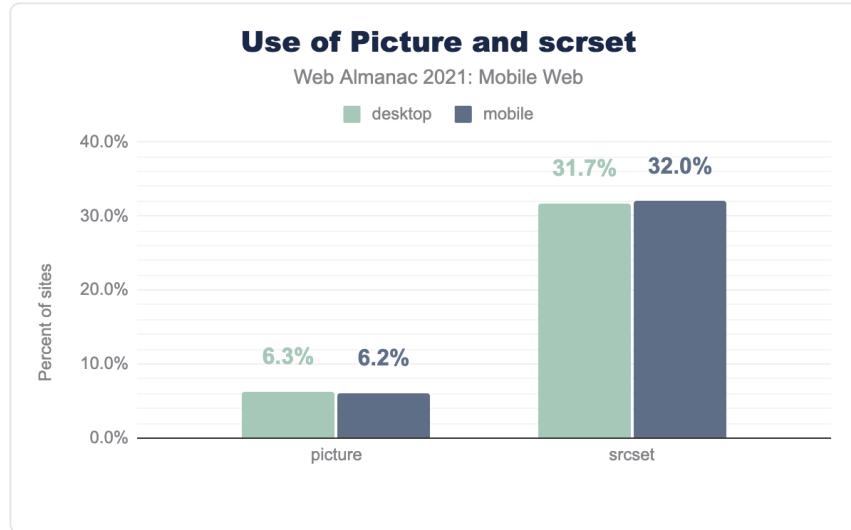


Figure 13.25. Use of `<picture>` and `srcset` to serve responsive images.

Just 6.2% of mobile page loads that included images used the `<picture>` element, slightly lower than desktop.

A healthier 32% of mobile page loads including images use the `srcset` attribute. It is worth mentioning here that this attribute can be used in both the `<picture>` element and the `<img>` element, so there's likely to be some crossover here.

## Lazy loading

Deferring, or lazy loading, images that aren't in the initial viewport is a good strategy to help resources be focused on loading things that are visible. The native lazy-load attribute, supported in Chrome, Opera, and from September 2021 Firefox for Android (source: caniuse.com<sup>616</sup>) allows this to happen without JavaScript workarounds.

18.4%

Figure 13.26. Mobile page loads that contained images used `loading="lazy"`

This is a big jump up from just 4.1% in 2020.

<sup>616</sup>. <https://caniuse.com/loading-lazy-attr>

Looking at the HTTP Archive's Native Image Lazy Loading Report<sup>617</sup>, uptake of using the `<img>` tag specifically shows the same, impressive growth.

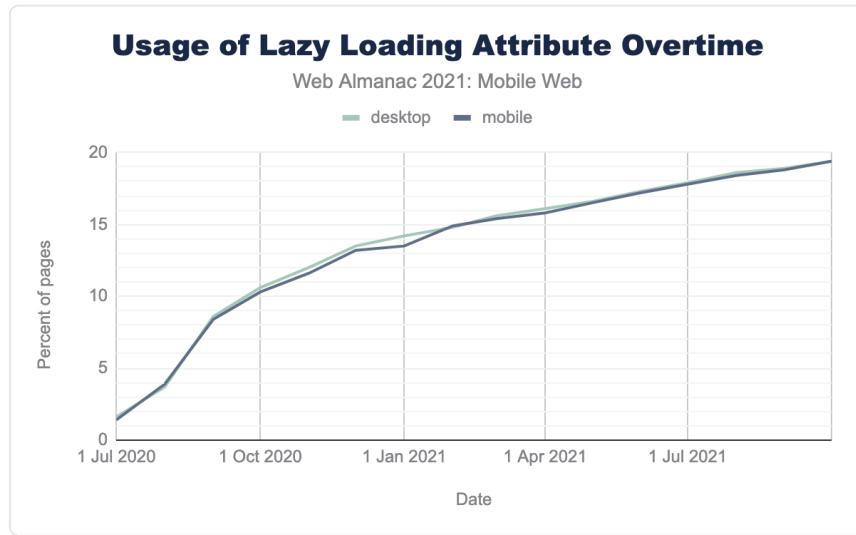


Figure 13.27. Usage of Lazy Loading attribute over time.

A driving factor in this growth can be attributed to the prevalence of WordPress (source: Rick Visconti on Twitter<sup>618</sup>). WordPress added support for native lazy-loading in version 5.5<sup>619</sup> which rolled out to the public on August 11th, 2020.

It's also worth mentioning that incorrectly used, Lazy Loading LCP Candidates<sup>620</sup> can harm performance. Making sure to apply `loading="lazy"` only to images below the fold is best practice.

## Image conclusions

It's disappointing to see that more mobile page loads this year had images that were not correctly sized. `<picture>` uptake remains low too, perhaps based on the complexity compared to the `<img>` element.

But great strides have been made in adoption of the `loading="lazy"` attribute, a huge jump in just one year.

617. <https://httparchive.org/reports/state-of-images#imgLazy>

618. [https://twitter.com/rick\\_visconti/status/1344380340153016321?s=20](https://twitter.com/rick_visconti/status/1344380340153016321?s=20)

619. <https://make.wordpress.org/core/2020/07/14/lazy-loading-images-in-5-5/>

620. <https://web.dev/lcp-lazy-loading/>

Images remain a vital part of the web, and that doesn't change for mobile users. If your site doesn't take advantage of some of the available approaches to serve mobile appropriate images, it's time to investigate this.

## Layout stability

With a generally smaller form factor, and limited screen real estate, unexpected shifting content can be particularly jarring on mobile devices.

Reading an article, only to have the paragraph you are on jump down the screen as an ad loads in above, or shift around as a font loads in and changes before your eyes, is an uncomfortable and negative experience.

### Cumulative Layout Shift

One of the Core Web Vitals, Cumulative Layout Shift<sup>621</sup> (CLS) is a metric designed to capture the impact of this kind of shifting of elements.

The metric is a calculation of impact fraction multiplied by distance fraction. The impact fraction is how much of the area of the screen is shifted and the distance fraction is how much of the screen it moved by.

A CLS score of 0.1 or under is considered good, under 0.25 considered indeed of improvement, and over that it's considered a poor experience

Smaller screen sizes are susceptible to greater shifts, at 360 x 640px, this example block causes a CLS score of 0.22

---

<sup>621</sup>. <https://web.dev/cls/>



Figure 13.28. Screen capture mock-up showing an ad causing CLS on a mobile sized screen.

At desktop screen sizes, the same element appearing leads to a CLS score of just 0.07.



Figure 13.29. Screen capture mock-up showing an ad causing CLS on a desktop sized screen.

The CrUX dataset shows that 62% of mobile page loads had a CLS of 0.1 or under:



Figure 13.30. CLS performance by device.

This is a big step over the 43% achieved last year, but direct comparison is hard, as the metric changed on the 1st of June 2021<sup>622</sup> to better capture the experience on long-lived pages, so some of this jump could be attributable to this.

## Response to user interaction

When a user interacts with a site, long delays from clicking on something, to something actually happening make a website or app feel sluggish and slow. This lag between input and the action happening is often down to heavy JavaScript processes blocking the main thread, leaving the browser unable to process the command the user issued until it had completed those processes.

Mobile devices are generally much lower powered than desktop and laptops, so the effect of this can be amplified.

### First Input Delay

First input delay<sup>623</sup> (FID) is the third Core Web Vital metric designed to capture this. It measures the time between the first interaction (a tap or a click on an element) until the browser can start processing that it has happened. It doesn't measure how long the process that tap may have

622. <https://web.dev/evolving-cls/>

623. <https://web.dev/fid/>

triggered takes.

A good FID score is 100 ms or under, a poor FID score is over 300 ms.

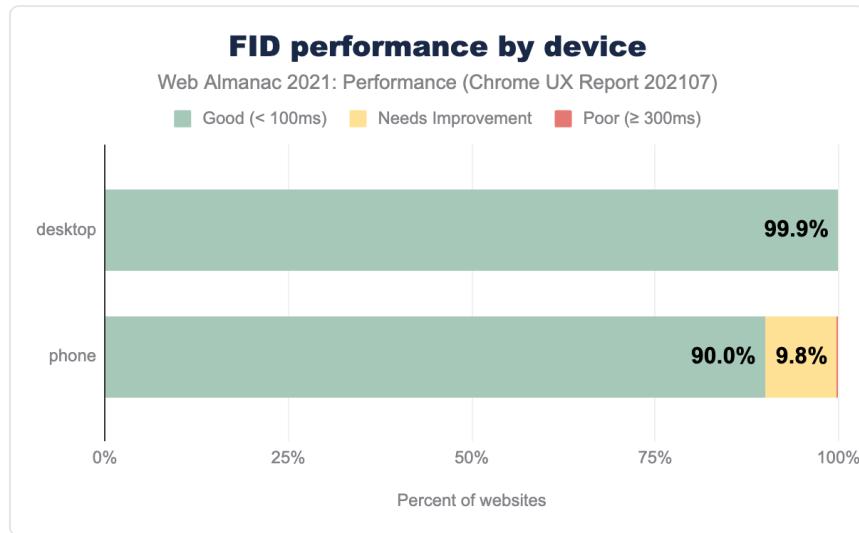


Figure 13.31. FID performance by device.

Encouragingly, 90% of mobile page loads in the CrUX dataset had a good FID score, up from 80% from 2020.

Efforts are being made to better capture responsiveness, with the Chrome Speed Metrics team sharing some plans and inviting feedback<sup>624</sup> on a new responsiveness metric.

If you are looking to learn more about Core Web Vitals in general, the Performance chapter has plenty of details about the Core Web Vitals.

## Service workers

Service workers<sup>625</sup> while not only applying to mobile devices do become uniquely useful in their ability to add offline capabilities, and better control of loading from caches to web apps, both features which are often more relevant to mobile users, who are more likely to encounter poor or total loss of connectivity.

14.8% of sites register a service worker, a sizeable uptake since 2020's 0.9%

<sup>624</sup>. <https://web.dev/responsiveness/>

<sup>625</sup>. [https://developer.mozilla.org/docs/Web/API/Service\\_Worker\\_API](https://developer.mozilla.org/docs/Web/API/Service_Worker_API)

To learn more about service workers and PWA (progressive web apps), visit the PWA chapter.

## Mobile performance conclusions

Overall, performance has taken a step forward over 2020, with a particularly strong improvement in layout stability.

There are some good, positive signs too in impressive usage growth in `loading="lazy"` and the uptake of service workers. The fact developers are embracing these is a positive sign that performance is being taken seriously.

It does however seem that improving Large Contentful Paint, and handing images are areas developers are struggling with more than other areas. Hopefully tooling and libraries like `next/image`<sup>626</sup> for the Next.js framework, and adoption by popular CMSs like WordPress will help developers overcome these pain points.

## Conclusion

In 2021, the perception of a distinct “mobile web” is outdated.

Across multiple data sources, it seems that the mobile is one of many ways a user can interact with digital content—and in fact comprises the majority of digital interactions.

For many users, mobile devices are their primary or only means of interacting with the web. Despite this, adoption of methodologies, performance strategies, accessibility principles and adoption of browser-supported features is low.

There has been great progress in some areas, most performance metrics are an improvement over 2020’s data. There do remain areas where there’s lots of room for growth too.

Accessibility remains an area where it would be great to see more effort and time spent, and image best practices still have some way to go.

With the continuing growth and size of the mobile user sector, for many industries it’s no longer a case of having to make a business case to support the mobile web, it is a case of fully embracing it and making use of the many tools and techniques available to a developer in 2021.

---

<sup>626</sup>. <https://nextjs.org/docs/api-reference/next/image>

## Authors



### Jamie Indigo

🐦 @Jammer\_Volts    Ⓛ fellowhuman1101    ☎ https://not-a-robot.com/

Jamie Indigo isn't a robot, but speaks bot. As a technical SEO consultant at Deepcrawl<sup>627</sup>, they study how search engines crawl, render, and index the web. They love to tame wild JavaScript frameworks and optimize rendering strategies. When not working, Jamie likes horror movies, graphic novels, and Dungeons & Dragons.



### Dave Smart

🐦 @davewsmart    Ⓛ dwsmart    ☎ https://tamethebots.com/

Dave Smart is a developer and technical search engine consultant at Tame the Bots<sup>628</sup>. They love building tools and experimenting with the modern web and can often be found at the front in a gig or two.



### Ashley Berman Hale

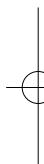
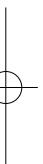
⌚ ashleyish

Ashley Berman Hale is a technical SEO and VP of professional services at Deepcrawl<sup>629</sup>. She is a mom to plants, animals, and tiny humans. Ashley plays in her local roller derby league and mentors upcoming SEOs.

627. <https://www.deepcrawl.com>

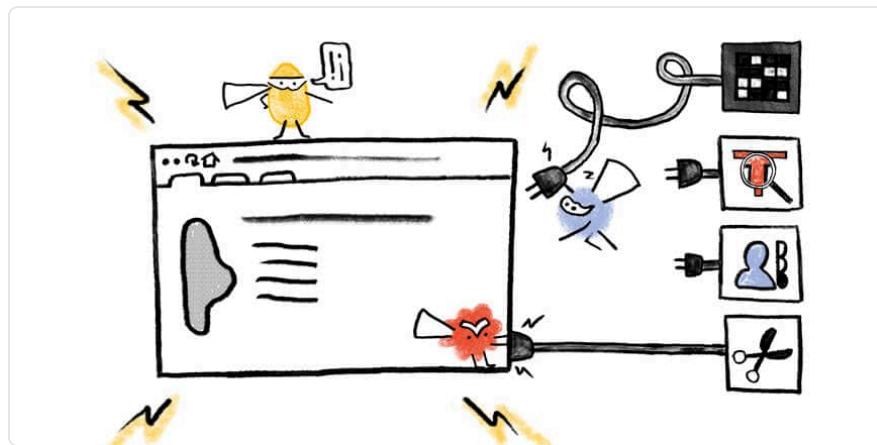
628. <https://tamethebots.com>

629. <https://www.deepcrawl.com>



# Part II Chapter 14

# Capabilities



**Written by Christian Liebel**

Reviewed by Thomas Steiner and Hemanth HM

Analyzed by Thomas Steiner

Edited by Barry Pollard

## Introduction

Capabilities are new web platform APIs that unlock entirely new use cases for web applications. Those new APIs are essential for Progressive Web Apps (PWA), a web-based application model. A PWA is a web app that users can install to their system. PWAs run even offline and launch quickly. To integrate with the underlying operating system, PWAs can only use web platform APIs. While browsers have already exposed some lower-level features to the web (e.g., geolocation<sup>630</sup>, gamepad<sup>631</sup>, or webcam<sup>632</sup> access), many APIs were still missing or were clumsy to use (e.g., file system or clipboard access).

630. [https://developer.mozilla.org/docs/Web/API/Geolocation\\_API](https://developer.mozilla.org/docs/Web/API/Geolocation_API)

631. [https://developer.mozilla.org/docs/Web/API/Gamepad\\_API](https://developer.mozilla.org/docs/Web/API/Gamepad_API)

632. <https://developer.mozilla.org/docs/Web/API/MediaDevices/getUserMedia>

## Project Fugu

The Capabilities Project<sup>633</sup> (codename Fugu) is a joint effort by Microsoft, Intel, Google, and other Chromium contributors. It tries to bridge the gap between platform-specific applications and web apps by designing and implementing new powerful web platform APIs in a secure and privacy-preserving manner (see also the Privacy chapter). As capabilities unlock more and more use cases, they lay the path for entire new application categories to finally make the shift to the web (e.g., IDEs, image editors, or office applications).

*Project Fugu is an effort to close gaps in the web's capabilities enabling new classes of applications to run on the web... APIs that Project Fugu is delivering enable new experiences on the web while preserving the web's core benefits of security, low-friction, and cross-platform delivery. All Project Fugu API proposals are made in the open and on the standards track.*

— Web Capabilities Team<sup>634</sup>

Over the last two years, the focus for the Fugu team has been on capabilities for desktop productivity applications and hardware-related APIs. This chapter briefly introduces several new capabilities and analyzes how many different desktop and mobile websites use them. As capabilities are particularly interesting for app-like websites, their relative usage is comparatively low. This is why absolute website numbers are used in this chapter. For each capability, there will be a demo website or app that makes use of it.

## Methodology

This chapter uses the HTTP Archive data set. For security reasons, some APIs require a user gesture (i.e., a click or keypress) to function. As the HTTP Archive crawler does not support detecting those APIs during runtime, the source code of the websites is parsed statically instead: For instance, the regular expression `/navigator\.share\s*\(/g` is matched against the website's source code to determine if it (potentially) makes use of the *Web Share API*.

This method is not perfectly accurate, as it doesn't measure the actual use of an API, and developers may invoke an API using a different syntax or work with minified code. However, this approach should provide a sufficiently good overview. You can find the exact regular expressions for the 30 supported capabilities in this source file<sup>635</sup>.

633. <https://www.chromium.org/teams/web-capabilities-fugu>

634. <https://www.chromium.org/teams/web-capabilities-fugu>

635. [https://github.com/HTTPArchive/legacy.httparchive.org/blob/master/custom\\_metrics/fugu-apis.js](https://github.com/HTTPArchive/legacy.httparchive.org/blob/master/custom_metrics/fugu-apis.js)

All usage data in this chapter is based on the July 2021 crawl. You can find the raw data in the Capabilities 2021 Results Sheet<sup>636</sup>.

For the two more commonly used APIs in this chapter, additional data from Chrome Platform Status is presented. This data shows how the API usage has changed over the last 12 months prior to the publication of this chapter.

## Status of the presented APIs

Please note that most of the APIs presented here are so-called *incubations*. Unless noted, they are not (yet) W3C Recommendations, i.e., official web standards. Instead, these APIs are being worked on in the Web Platform Incubator Community Group (WICG), where browser vendors and developers can discuss new features.

Some APIs have already shipped in several browsers; others are only available on Chromium-based ones. These browsers include Google Chrome, Microsoft Edge, Opera, Brave, or Samsung Internet. Please note that vendors of Chromium-based browsers can choose to disable specific capabilities, so not all APIs may be available in all browsers based on Chromium. Some capabilities may also only be available after activating a flag in the browser settings.

## Async Clipboard API

The Async Clipboard API allows you to read and write data from or to the clipboard. Due to its asynchronous nature, it enables use cases like scaling down an image while pasting it—all without blocking the UI. It replaces less capable APIs like `document.execCommand()` that were previously used to interact with the clipboard.

### Write access

The Async Clipboard API offers two methods to copy data to the clipboard: The shorthand method `writeText()` takes plain text as an argument which the browser then copies to the clipboard. The `write()` method takes an array of clipboard items that could contain arbitrary data. Browsers can decide to only implement certain data formats. The Clipboard API specification specifies a list of mandatory data types<sup>637</sup> browsers must support as a minimum, including plain text, HTML, URI lists, and PNG images.

636. <https://docs.google.com/spreadsheets/d/1b4moteB9ElYkH1Ln9qf1tnU-E4N2UQ87uyWyDKw/>

637. <https://www.w3.org/TR/clipboard-apis/#mandatory-data-types-x>

```

await navigator.clipboard.writeText('hello world');

const blob = new Blob(['hello world'], { type: 'text/plain' });
await navigator.clipboard.write([
  new ClipboardItem({
    [blob.type]: blob,
  }),
]);

```

## Read access

Similar to copying data to the clipboard, there are two methods to paste data back from the clipboard: First, another shorthand method called `readText()` that returns plain text from the clipboard. Using the `read()` method, you access all items in the clipboard in the data formats supported by the browser.

```

const item = await navigator.clipboard.readText();
const items = await navigator.clipboard.read();

```

The browser may show a permission prompt or a different UI for privacy reasons before granting the website access to the clipboard contents. The Async Clipboard API is available in Chrome, Edge, and Safari (current browser support for the Async Clipboard API<sup>638</sup>). Firefox only supports the `writeText()` method.

560,359

*Figure 14.1. Desktop websites using the Async Clipboard API.*

With 560,359 (8.91%) desktop and 618,062 (8.25%) mobile sites, the Async Clipboard API (`writeText()` method) is one of the most used Fugu APIs. The `write()` method is used on 1,180 desktop and 1,227 mobile sites. As an example, the commercial website Clipping Magic<sup>639</sup> allows you to remove the background of an image with the help of an AI algorithm. Just paste an

638. <https://caniuse.com/async-clipboard>

639. <https://clippingmagic.com/>

image from the clipboard, and the website will remove its background.

The high usage of this API is probably related to a script that is included with embedded YouTube videos. The `writeText()` method is called when the user clicks the “copy link” button in the video player.

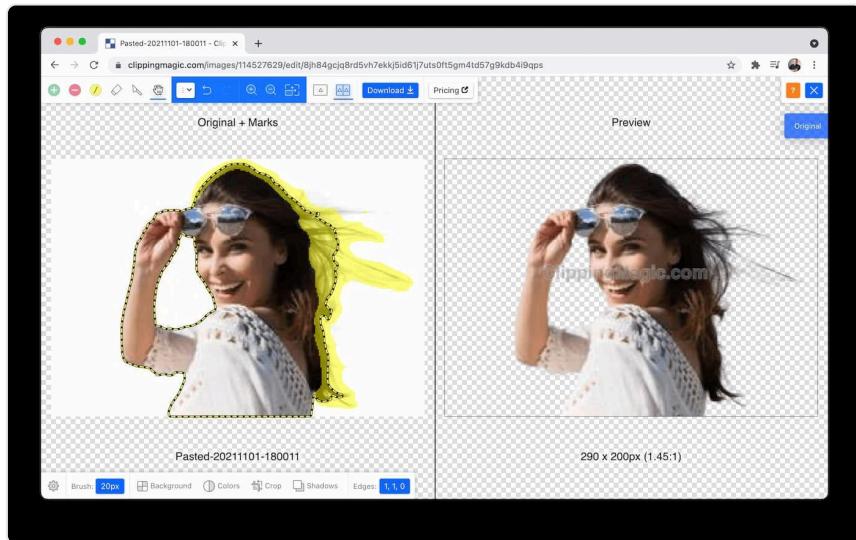


Figure 14.2. Clipping Magic uses artificial intelligence to remove the background of images pasted via the Async Clipboard API.

In recent months, the use of the API has increased sharply at a low level. While the `read()` method was active on only 0.00032 percent of all page loads in November 2020, usage increased exponentially to 0.002921 percent by October 2021. The `write()` method increased from 0.000674 to 0.001601 percent in the same period.



Figure 14.3. Percentage of page loads in Chrome using Async Clipboard API.  
(Sources: Async Clipboard Read<sup>640</sup>, Async Clipboard Write<sup>641</sup>)

## File System Access API

The next productivity-related API is the *File System Access API*. Web apps could already deal with files<sup>642</sup>: `<input type="file">` allows the user to open one or more files via a file picker. Also, they could already save files to the Downloads folder via `<a download>`. The File System Access API adds support for additional use cases: Opening and modifying directories, saving files to a location specified by the user, and overwriting files that were opened by them. It is also possible to persist file handles to `IndexedDB` to allow for continued (permission-gated) access, even after a page reload. In particular, the API does not grant random access to the file system and certain system folders are blocked by default.

### Write access

When calling the `showSaveFilePicker()` method on the global `window` object, the browser will show the operating system's file picker. The method takes an optional options object where you can specify which file types are allowed for saving (`types`, default: all types), and whether the user can disable this filter via an "accept all" option

640. <https://chromestatus.com/metrics/feature/timeline/popularity/2369>

641. <https://chromestatus.com/metrics/feature/timeline/popularity/2370>

642. <https://web.dev/browser-fs-access/#the-traditional-way-of-dealing-with-files>

```
(excludeAcceptAllOption, default: false).
```

When the user successfully picks a file from the local file system, you will receive its handle. With the help of the `createWritable()` method on the handle, you can access a stream writer. In the following example, this writer writes the text `hello world` to the file and closes it afterward.

```
const handle = await window.showSaveFilePicker({
  types: [{  
    description: 'PNG files',  
    accept: { 'image/png': ['.png'] }  
  }],
  excludeAcceptAllOption: true
});  
const writable = await handle.createWritable();  
await writable.write('hello world');  
await writable.close();
```

## Read access

To show an open file picker, call the `showOpenFilePicker()` method on the global `window` object. This method also takes an optional options object with the same properties from above (`types`, `excludeAcceptAllOption`). Additionally, you can specify if the user can select one or multiple files (`multiple`, `default: false`).

As the user could potentially select more than one file, you will receive an array of file handles. Using the array destructuring expression `[handle]`, you will receive the handle of the first selected file as the first element in the array. By calling the `getFile()` method on the file handle, you will receive a `File` object which gives you access to the file's binary data. By calling the `text()` method, you will receive the plain text from the opened file.

```
const [handle] = await window.showOpenFilePicker({  
  multiple: false  
});  
const blob = await handle.getFile();  
const text = await blob.text();
```

```
console.log(text);
```

## Opening directories

Finally, the API allows web apps (e.g., integrated development environments) to get a handle for an entire directory. Using this handle, you can create, update, or delete existing files or folders within the opened directory. This time, the method is called `showDirectoryPicker()`:

```
const handle = await window.showDirectoryPicker();
```

The File System Access API is only available on Chromium-based browsers and desktop systems (current browser support for the File System Access API<sup>643</sup>). Fortunately, the web platform offers the aforementioned fallback approaches to provide similar functionality on mobile devices and other browsers. Developers can use the Google-developed library `browser-fs-access`<sup>644</sup> that uses the File System Access API if present and otherwise falls back to the alternative implementation.

# 29

*Figure 14.4. Desktop websites using the File System Access API.*

Out of all 6,286,373 desktop and 7,491,840 mobile websites in the HTTP Archive, the File System Access API is used on 29 desktop and 23 mobile sites. Examples for those sites are the image editor Excalidraw<sup>645</sup>, which allows you to sketch diagrams in a hand-drawn look and save them to the disk. Another example is CorelDRAW.app<sup>646</sup>, a web version of the image editing software CorelDRAW.

643. <https://caniuse.com/native-filesystem-api>  
644. <https://github.com/GoogleChromeLabs/browser-fs-access>  
645. <https://excalidraw.com/>  
646. <https://coreldraw.app/>

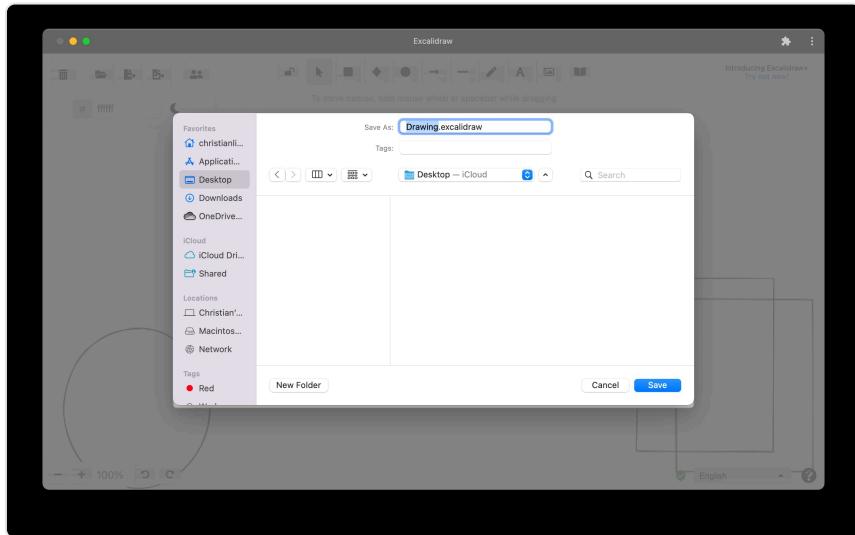


Figure 14.5. The Excalidraw PWA uses the File System Access API to save images to the local file system via the built-in save dialog.

## Web Share API

The *Web Share API* allows you to share text, a URL, or files from a website or web application with other applications, e.g., mail clients or messengers. To do so, call the `navigator.share()` method. It takes an object with the data to share with another application. The browser then opens the built-in share sheet, where the user can select the target application from. The method returns a promise that resolves in case the content was successfully shared; otherwise, it will be rejected.

```
await navigator.share({
  files: picturesArray,
  title: 'Holiday pictures',
  text: 'Our holiday in the French Alps'
})
```

The Web Share API is supported by Safari on iOS and macOS, and Chrome and Edge on

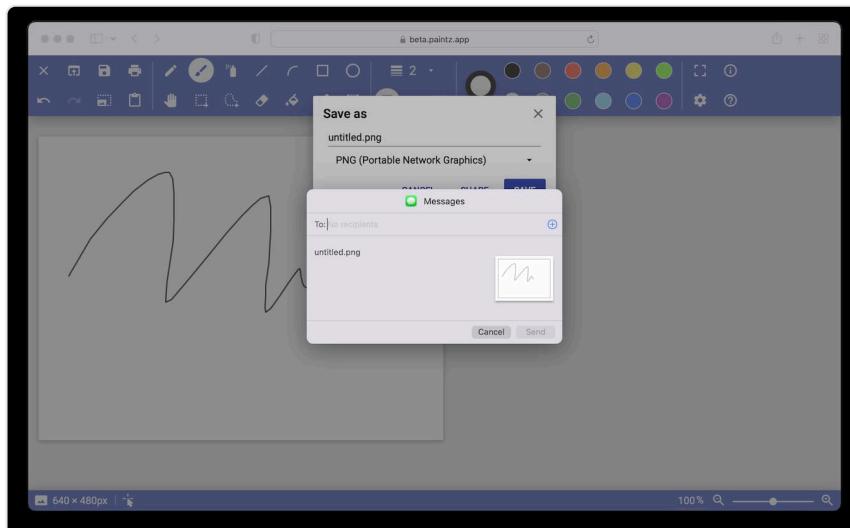
Windows and Chrome OS (current browser support for the Web Share API<sup>647</sup>). It's currently a Working Draft<sup>648</sup> at the Web Applications Working Group. This is one of the first stages of the track to becoming a W3C Recommendation.

# 566,049

*Figure 14.6. Desktop websites using the Web Share API.*

With 566,049 (9.00%) desktop and 642,507 (8.58%) mobile sites, the Web Share API is the most used Fugu API. For example, the beta version of the PaintZ app<sup>649</sup> allows you to share a drawing with another locally installed application via the save dialog.

The high usage of this API is probably related to a script that is included with embedded YouTube videos. If the Web Share API is available on the device, it is executed when the user clicks the "Share" button in the video player.



*Figure 14.7. The beta version of PaintZ uses the Web Share API to share drawings with local applications.*

In recent months, the overall use of the Web Share API has increased: The Chrome Platform Status data shows a rather linear growth in the period from November 2020, where the API

647. <https://caniuse.com/web-share>

648. <https://www.w3.org/TR/web-share/>

649. <https://beta.paintz.app/>

was called on 0.0097% of all page loads, to 0.0136% in October 2021.

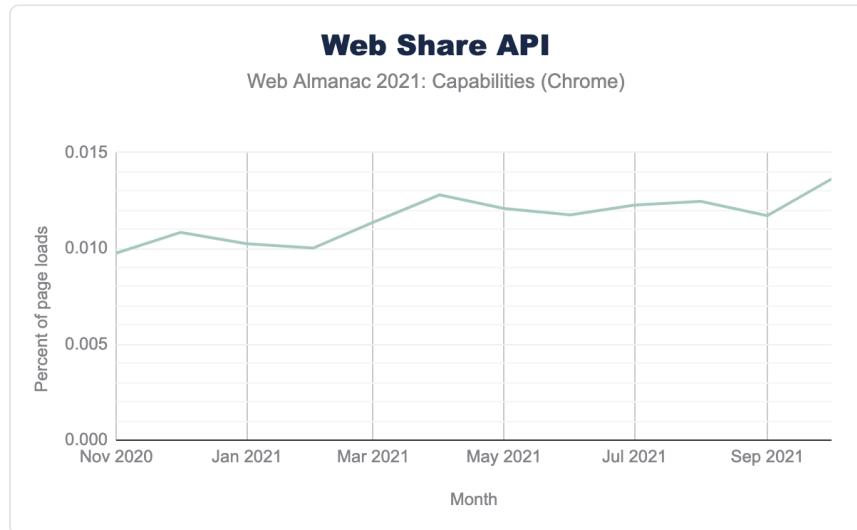


Figure 14.8. Percentage of page loads in Chrome using Web Share API. (Source<sup>650</sup>)

## URL Handlers and Declarative Link Capturing

The last two productivity-related capabilities described in this chapter are *URL Handlers* and *Declarative Link Capturing*, additional methods for even deeper integration with the operating system.

### URL Handling

With the help of URL Handling<sup>651</sup>, PWAs can register themselves as handlers for certain URL schemes upon installation, e.g., for `https://*.example.com`. When the user opens a URL that matches this scheme, the installed PWA will open instead of a new browser tab. URL Handling is an extension of the *Web Application Manifest*, a file that contains metadata for web applications<sup>652</sup>. To register for URL schemes, you have to add the `url_handlers` property to your manifest. This property takes an array containing objects with an `origin` property.

650. <https://chromestatus.com/metrics/feature/timeline/popularity/1501>

651. <https://web.dev/pwa-url-handler/>

652. <https://developer.mozilla.org/docs/Web/Manifest>

```
{
  "url_handlers": [
    {
      "origin": "https://*.example.com"
    }
  ]
}
```

If you want to register for origins other than your web app's origin, you need to verify your ownership of them<sup>653</sup>. The capability is at a relatively early stage: it's only supported on Chrome and Edge on the desktop. URL Handling is currently available as an Origin Trial<sup>654</sup>. This means that the capability is not generally available yet. Instead, developers need to opt-in to using this experimental API by registering for an Origin Trial token first and deliver this token along with their website to use this capability. You can find more information in the Origin Trials Guide for Web Developers<sup>655</sup>.

# 44

*Figure 14.9. Desktop websites use URL Handling.*

44 desktop and 41 mobile websites make use of URL Handling. For example, the Pinterest PWA registers itself as a URL handler for the different Pinterest origins (e.g., `*.pinterest.com` and `*.pinterest.de`) on installation.

## Declarative Link Capturing

With the help of Declarative Link Capturing<sup>656</sup>, you can further control how PWAs should behave when the user opens them. For instance, an office application may want to open another window for a new document, while a music player wants to keep its single window open. Therefore, Declarative Link Capturing defines three different modes:

1. `none` does not capture the link at all (the default)
2. `new-client` opens a new window for the PWA
3. `existing-client-navigate` navigates an existing client to the new URL or opens a new window if no client exists

653. <https://web.dev/pwa-url-handler/#the-web-app-origin-association-file>

654. <https://developer.chrome.com/blog/origin-trials/>

655. <https://github.com/GoogleChrome/OriginTrials/blob/gh-pages/developer-guide.md>

656. <https://web.dev/declarative-link-capturing/>

Declarative Link Capturing also is an extension of the Web Application Manifest. To use it, you need to add the `capture_links` property to your manifest. This property takes a string or an array of strings matching the three modes from above. If you use an array, the browser will fall back to the next entry if it doesn't support a particular mode.

```
{
  "capture_links": [
    "existing-client-navigate",
    "new-client",
    "none"
  ]
}
```

# 36

*Figure 14.10. Desktop websites use Declarative Link Capturing.*

This capability is at an early stage as well. It is only supported on Chrome OS. Currently, 36 desktop sites and 11 mobile sites use this capability, for example, Periodex<sup>657</sup>, a PWA showing the periodic table of elements. This app uses the `capture_links` configuration as shown in the listing above meaning that, if supported, the browser should reuse the existing window, otherwise, open a new one, and if that's not supported, it should behave as normal.

## Hardware APIs

The next set of capabilities focuses on hardware-related APIs. In Chromium-based browsers, there are many APIs to access hardware interfaces, including but not limited to USB, Bluetooth, and serial devices. Furthermore, the Generic Sensor API allows you to read from device sensors. All capabilities discussed in this section are only available on Chromium-based browsers and on systems where the respective hardware interface or sensor is present.

### Web USB API

The *Web USB API* allows developers to access USB devices without any drivers or third-party

657. <https://periodex.co/>

applications. For instance, this capability is interesting for firmware updates that developers otherwise would have to implement as separate platform-specific apps for different platforms. You need to call the `navigator.usb.requestDevice()` method to access USB devices. It takes an object which defines filters for the list of all connected USB devices. You need to specify the `vendorId` at least. The browser shows a device picker where the user can choose a matching device. From there, you can begin a device session.

```
try {
  const device = await navigator.usb.requestDevice({
    filters: [{ vendorId: 0x8086 }]
  });
  console.log(device.productName);
  console.log(device.manufacturerName);
} catch (err) {
  console.log(err);
}
```

# 182

Figure 14.11. Desktop websites use Web USB.

The API has been generally available on Chromium-based browsers since version 61 (current browser support for the Web USB API<sup>658</sup>). 182 desktop and 155 mobile sites use this API, for example, the PWA Vysor<sup>659</sup> that allows you to mirror the screen of an Android or iOS device—all without installing any additional software on your computer.

658. <https://caniuse.com/webusb>  
659. <https://app.vysor.io/#/>

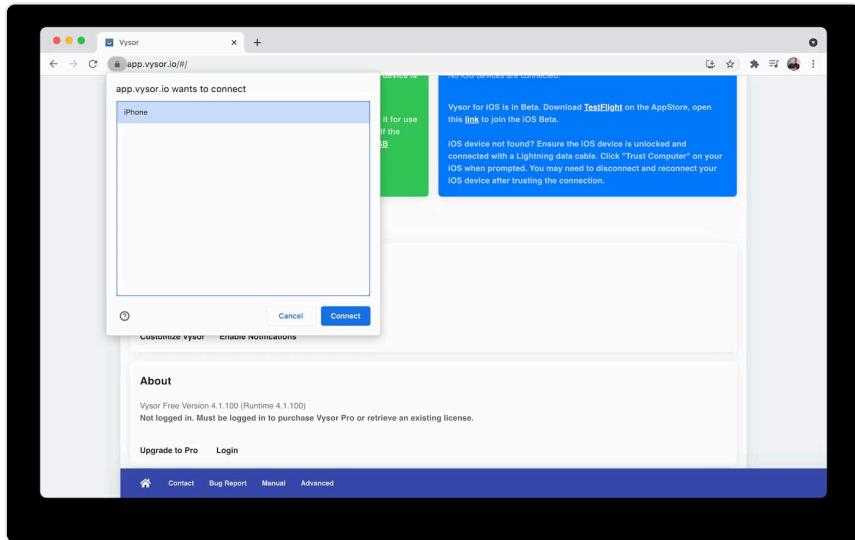


Figure 14.12. The Vysor PWA uses Web USB to connect to USB devices and project their screen contents onto the desktop.

## Web Bluetooth API

The *Web Bluetooth API* allows you to communicate with nearby Bluetooth Low Energy devices using the Generic Attribute Profile (GATT)<sup>660</sup>. To find a matching device, call the `navigator.bluetooth.requestDevice()` method. In the following example, the list of Bluetooth devices is filtered by whether they offer a battery service or not. The browser shows a device picker where the user can choose a Bluetooth device. Afterward, you can connect to the remote device and gather the data.

```
try {
  const device = await navigator.bluetooth.requestDevice({
    filters: [{ services: ['battery_service'] }]
  });
  console.log(device.name);
} catch (err) {
  console.log(err);
```

<sup>660</sup> <https://www.bluetooth.com/bluetooth-resources/intro-to-bluetooth-gap-gatt/>

}

# 71

Figure 14.13. Desktop websites using the Web Bluetooth API.

The API is generally available on Chromium-based browsers on Chrome OS, Android, macOS, and Windows starting from version 56 (current browser support for the Web Bluetooth API<sup>661</sup>). On Linux, the API is provided behind a flag. 71 desktop and 45 mobile sites make use of this capability. For instance, the Brewfather<sup>662</sup> PWA targeted at home brewers allows them to send a beer recipe wirelessly over to a Bluetooth-enabled brewing system. Again, all without installing any third-party software.



Figure 14.14. The Brewfather app uses Web Bluetooth to send recipes to a brew controller.

## Web Serial API

The Web Serial API allows you to connect with serial devices such as microcontrollers. To do so,

661. <https://caniuse.com/web-bluetooth>

662. <https://web.brewfather.app/>

call the `navigator.serial.requestPort()` method. You can optionally pass in a method to filter the device list. The browser shows a device picker where the user can choose a device. Next, you can open the connection by calling the port's `open()` method.

```
try {  
    const port = await navigator.serial.requestPort();  
    await port.open({ baudRate: 9600 });  
} catch (err) {  
    console.log(err);  
}
```

# 15

*Figure 14.15. Desktop websites using the Web Serial API.*

This capability is relatively new, as it shipped with Chromium 89 in March 2021 (current browser support for the Web Serial API<sup>663</sup>). Currently, 15 desktop and 14 mobile sites use the Web Serial API, including the Duino App<sup>664</sup> that allows you to develop programs for Arduino and ESP microcontrollers right in your browser. They are compiled on a remote server and then uploaded to a connected board via the Web Serial API.

663. <https://caniuse.com/web-serial>  
664. <https://duino.app/>

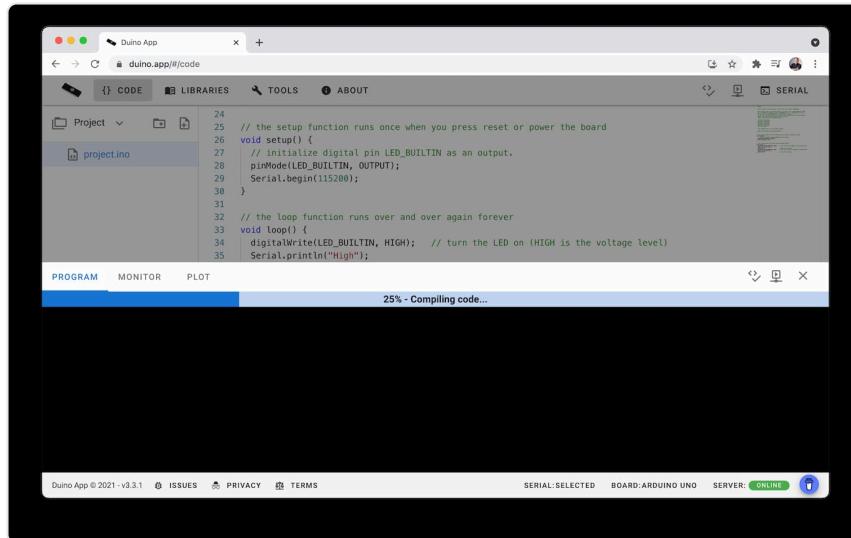


Figure 14.16. The Duino app is a web-based IDE that uses Web Serial to upload programs to Arduino microcontrollers.

## Generic Sensor API

Finally, the *Generic Sensor API* allows you to read sensor data from the device’s sensors, such as the accelerometer, gyroscope, or orientation sensor. To access a sensor, you create a new instance of a sensor class, e.g., `Accelerometer`. The constructor takes a configuration object with the requested frequency. By attaching to the `onreading` and `onerror` events, you can get notified for updated sensor values, or errors respectively. Finally, you need to start the reading by calling the `start()` method.

```

try {
  const accelerometer = new Accelerometer({ frequency: 10 });
  accelerometer.onerror = (event) => {
    console.log(event);
  };
  accelerometer.onreading = (e) => {
    console.log(e);
  };
  accelerometer.start();
}
  
```

```

} catch (err) {
  console.log(err);
}

```

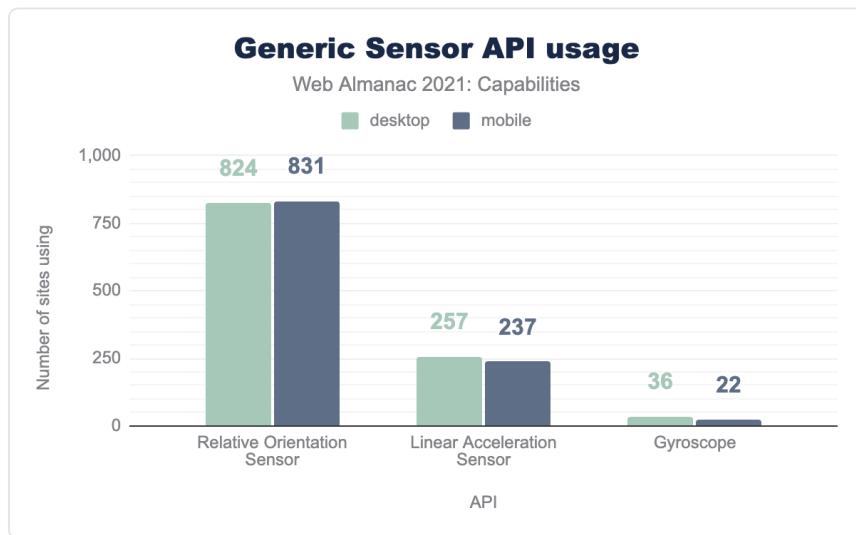


Figure 14.17. Usage of Generic Sensor APIs on desktop and mobile websites.

The capability is supported by Chromium browsers starting from version 67 (current browser support for the Generic Sensor API<sup>665</sup>). The relative orientation sensor is used by 824 desktop and 831 mobile sites, the linear acceleration sensor by 257 desktop and 237 mobile sites, and the gyroscope by 36 desktop and 22 mobile sites. An example application that uses all three of them is VDO.Ninja<sup>666</sup>, the former OBS Ninja. This software allows you to remotely connect with video broadcasting software such as OBS. The app allows the connected broadcasting software to read sensor data from the device. For example, to capture a smartphone's movements when streaming virtual reality content. Fugu contributor Intel provides additional demos for the Generic Sensor API<sup>667</sup>.

665. [https://caniuse.com/mdn-api\\_sensor](https://caniuse.com/mdn-api_sensor)

666. <https://obs.ninja/>

667. <https://intel.github.io/generic-sensor-demos/>



Figure 14.18. The Generic Sensor API can be used to rotate 3D models according to the orientation of the device.

## Sites using the most capabilities

The analysis also identified the websites using the most capabilities from the HTTP Archive data set. The detection script is capable of identifying 30 Fugu APIs in total. So, let's give an award to the websites that use the most Fugu APIs. The excitement is building!

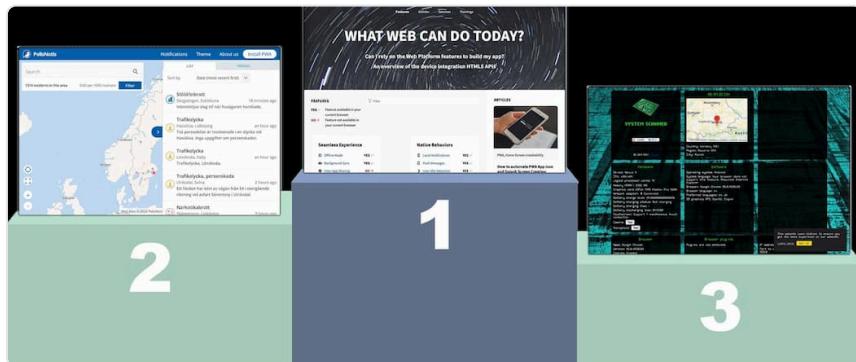


Figure 14.19. The three websites that use the most Fugu APIs.

1. The first place goes to [whatwebcando.today](https://whatwebcando.today/)<sup>668</sup>, which uses 28 capabilities. It showcases different HTML5 device integration APIs by providing a live demo for every capability. Naturally, the number of used APIs is very high. In the result set, a similar site called [whatpwacando.today](https://whatpwacando.today/)<sup>669</sup> showcases PWA capabilities and uses eight APIs.
2. The runner-up is the PolisNotis<sup>670</sup> PWA which shows police notices in Sweden. It uses ten APIs, including the Declarative Link Capturing API to define that the PWA should always open a new window when clicking a PWA-related link. The Web Share API is used in the source code, but the sharing functionality is not exposed to the UI. The app also uses the Badging API to alert the user via the app icon if there is a new notice.
3. Closely followed in third place is the website System Scanner<sup>671</sup>, that uses nine APIs: It shows an overview of the system information exposed by the browser, including sensor information provided by the Generic Sensor API.
4. Eight sites use eight Fugu APIs: One of them is the aforementioned Excalidraw<sup>672</sup>, an online drawing tool for creating drawings in a hand-drawn style. As a traditional productivity app, it benefits from the new capabilities.

Some websites from the result set are Internet forums based on Discourse<sup>673</sup>. This forum software supports a total of eight Fugu APIs. Discourse-based forums are installable and support, among others, the Badging API to show the number of unread notifications.

The results also include sites that aren't proactively using the APIs. For example, some sites ship library code that could theoretically access the capabilities. Some sites check for the presence

668. <https://whatwebcando.today/>

669. <https://whatpwacando.today/>

670. <https://polisnotis.se/>

671. <https://system-scanner.net/>

672. <https://excalidraw.com/>

673. <https://www.discourse.org/>

of Fugu APIs to determine the user's browser.

## Conclusion

Capabilities help move the web forward by unlocking more and more use cases for developers. As this chapter shows, developers use the new web platform APIs to build powerful applications. In contrast to their platform-specific counterparts, those applications don't necessarily need to be installed to the system and don't require any additional third-party runtimes or plugins to work. They run on any platform that can run a powerful browser.

One example of this concept working is Visual Studio Code. This application has always been web-based, but it still relied on platform-specific application wrappers like Electron. Thanks to capabilities like the File System Access API, Microsoft was able to release the application as a browser application (`vscode.dev`<sup>674</sup>) in October 2021. Almost all features work here, except debugging or terminal access since there is no capability for this (yet!).

Another example is Adobe Photoshop<sup>675</sup>, which was also released as a web application<sup>676</sup> in October 2021. Photoshop uses several of the capabilities presented here, as well as WebAssembly, to migrate existing code to the web. Its vector-based counterpart Illustrator is currently available as a closed beta and will be released at a later date. While the first editions will still have a limited feature set, Adobe has already announced that it won't stop there, but that further expansion to the web is planned<sup>677</sup>.

Thus, the Capabilities project paves the way for entire categories of applications to finally migrate to the web.

### Author



#### Christian Liebel

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

Christian Liebel is a consultant at Thinktecture<sup>678</sup>, supporting clients from various business areas in implementing great 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.

674. <https://vscode.dev>

675. <https://photoshop.adobe.com>

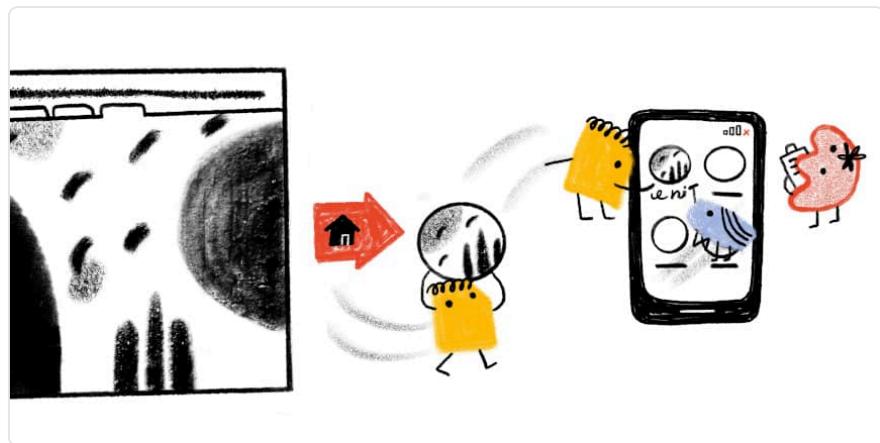
676. <https://web.dev/ps-on-the-web/>

677. <https://web.dev/ps-on-the-web/#what-s-next-for-adobe-on-the-web>

678. <https://thinktecture.com>

# Part II Chapter 15

# PWA



**Written by Demian Renzulli**

Reviewed by Barry Pollard, Maxim Salnikov, Jeff Posnick, André Cipriani Bandarra, Kai Hollberg, Hemanth HM, Pascal Schilp, and Adriana Jara

Analyzed by Barry Pollard and Demian Renzulli

Edited by Rick Viscomi

## Introduction

Six years have passed since Frances Berriman<sup>679</sup> and Alex Russell<sup>680</sup> coined the term “Progressive Web App” (PWA)<sup>681</sup>, which represented their vision for web apps that can be just as immersive as native apps. The following attributes were listed to distinguish these types of experiences from traditional websites:

- Responsive
- Progressively enhanced with service workers
- Having app-like interactions

679. <https://twitter.com/phae>

680. <https://twitter.com/slightlylate>

681. <https://infrequently.org/2015/06/progressive-apps-escaping-tabs-without-losing-our-soul/>

- Fresh
- Safe
- Discoverable
- Re-engageable
- Linkable

Over the last several years, the web platform has continued to evolve, reducing the gap between web apps and OS-specific experiences, and allowing developers to provide users with richer capabilities and new ways to stay engaged.

Despite that, it's still difficult to draw a clear line between what is a PWA or not; some experts might give more importance to creating an “appy” experience, characteristic of the shell and content application model<sup>682</sup>, while others focus more on certain components and behaviors, like having a service worker and a web app manifest, providing an offline experience, or other advanced functionalities.

In this year's PWA chapter, we'll focus on all the measurable aspects of a PWA: usage of service workers and its related APIs, web app manifests, and the most popular libraries and tools to build PWAs. A PWA can use all or some of these functionalities. We'll look at the level of adoption of each component and API to get an idea of the level of penetration of these technologies in the web ecosystem.

*Note: This chapter will focus mostly on service worker related APIs in common use. For more cutting-edge APIs, make sure to check out the Capabilities chapter.*

## Service workers

Service workers<sup>683</sup> (introduced in December 2014) are one of the core components of a PWA. They act as a network proxy and allow for features like offline, push notifications, and background processing, which are characteristic of “app-like” experiences.

It took some time for service workers to become widely adopted, but today they are supported by most major browsers<sup>684</sup>. However, this doesn't mean that all service worker features work across browsers. For example, while most of the core functionalities like network proxying are available, APIs like Push<sup>685</sup> are not yet available in WebKit<sup>686</sup>.

---

682. <https://developers.google.com/web/fundamentals/architecture/app-shell>

683. [https://developer.mozilla.org/docs/Web/API/Service\\_Worker\\_API](https://developer.mozilla.org/docs/Web/API/Service_Worker_API)

684. <https://caniuse.com/serviceworkers>

685. <https://caniuse.com/push-api>

## Service workers usage

We estimate that between 1.22% to 3.22% of sites use service workers in 2021, depending on the type of measurement used. This year we have decided to take the 3.22% as the closest approximation—for reasons we'll explain next.

# 3.22%

*Figure 15.1. Percent of mobile sites that use service workers.*

Measuring whether a service worker is used is not as simple as might seem. For example, Lighthouse detects 1.5%, however it adds some extra checks in that definition<sup>686</sup> rather than just service worker usage so could be seen as a lower bound. Chrome itself measures 1.22% sites using service workers<sup>687</sup>, which is strangely less than Lighthouse for reasons that we have not been able to ascertain.

For this year's PWA chapter, we've updated our measurement techniques by creating a new set of metrics<sup>688</sup>. For example, we're now using heuristics that check for several service worker characteristics, like having service worker registration<sup>689</sup> calls and use of service worker specific methods, libraries, and events.

From the data we gathered, we can see that about 3.05% of desktop sites and 3.22% of mobile sites use service workers features, which suggests that service worker usage might be higher than measured in last year's chapter<sup>690</sup> (0.88% in desktop and 0.87% in mobile).

One might think that having a little more than 3% of sites registering a service worker in mobile and desktop is a low number, but how does this translate to web traffic?

Chrome Platform Status<sup>691</sup> provides usage statistics obtained from the Chrome browser. According to those stats, service workers control 19.26% of page loads in July 2021<sup>692</sup>. Compared to last year's measurement of 16.6%<sup>693</sup>, this represents a yearly growth of 12% in page loads controlled by service workers.

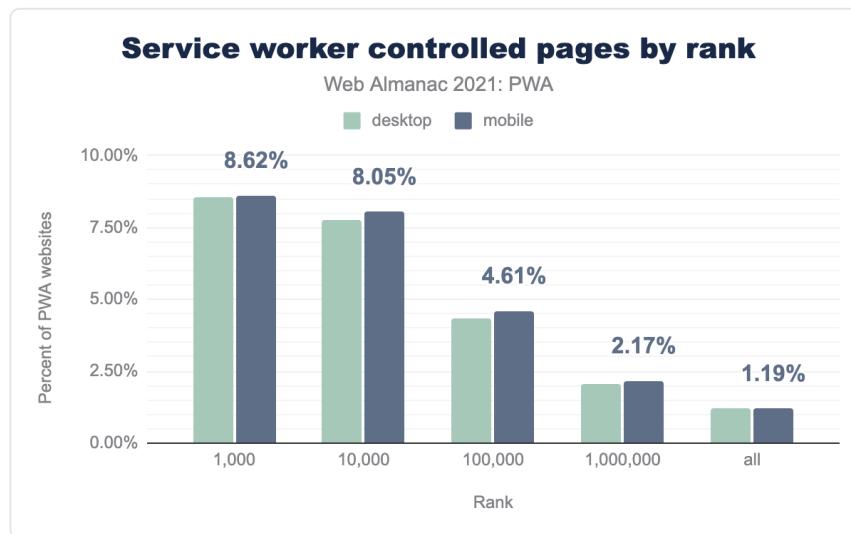
---

686. <https://web.dev/service-worker>  
 687. <https://httparchive.org/reports/progressive-web-apps#swControlledPages>  
 688. [https://github.com/HTTPArchive/legacy.httparchive.org/blob/master/custom\\_metrics/pwajs](https://github.com/HTTPArchive/legacy.httparchive.org/blob/master/custom_metrics/pwajs)  
 689. <https://developer.mozilla.org/docs/Web/API/ServiceWorkerRegistration>  
 690. <https://almanac.httparchive.org/en/2020/pwa#service-worker-usage>  
 691. <https://www.chromestatus.com/features>  
 692. <https://www.chromestatus.com/metrics/feature/timeline/popularity/990>  
 693. <https://almanac.httparchive.org/en/2020/pwa#service-worker-usage>

# 19.26%

*Figure 15.2. Percent of page views on a page that registers a service worker. (Source: Chrome Platform Status<sup>694</sup>)*

And how can we explain that approximately 3% of sites represent around 19% of the web traffic? Intuitively, one might think that high traffic websites have more reasons to adopt service workers. Having a larger user base means that users might arrive at the site from a variety of devices and connectivities, so the incentives to adopt APIs that provide performance benefits and reliability are higher. Also, these companies often have native apps, so there are more reasons to bridge the UX gap between platforms, by implementing advanced capabilities via service workers. The following data helps us prove that assumption:



*Figure 15.3. Service worker controlled pages by rank.*

When measuring the top 1,000 sites, 8.62% of them use service workers. As we broaden the number of sites under analysis, the overall percentage starts to decrease. This indicates that the most popular sites are more prone to use features like service workers and advanced capabilities.

694. <https://www.chromestatus.com/metrics/feature/timeline/popularity/990>

## Service worker features

In this section, we'll analyze the adoption of various service worker features (events<sup>695</sup>, properties<sup>696</sup>, methods<sup>697</sup>) for most common PWA tasks (offline, push notifications, background processing, etc.).

### Service worker events

The `ServiceWorkerGlobalScope`<sup>698</sup> interface represents the global execution context of a service worker and is governed by different events<sup>699</sup>. One can listen to them in two ways: via event listeners or service worker properties.

For example, here are two ways of listening to the `install` event in a service worker:

```
// Via event listener:  
this.addEventListener('install', function(event) {  
  // ...  
});  
  
// Via properties:  
this.oninstall = function(event) {  
  // ...  
};
```

We have measured and combined both ways of implementing event listeners and obtained the following stats:

695. <https://developer.mozilla.org/docs/Web/API/ServiceWorkerGlobalScope#events>

696. <https://developer.mozilla.org/docs/Web/API/ServiceWorkerGlobalScope#properties>

697. <https://developer.mozilla.org/docs/Web/API/ServiceWorkerGlobalScope#methods>

698. <https://developer.mozilla.org/docs/Web/API/ServiceWorkerGlobalScope>

699. <https://developer.mozilla.org/docs/Web/API/ServiceWorkerGlobalScope#events>



Figure 15.4. Most used service worker events.

We can divide these events results into 3 subcategories:

- Lifecycle events
- Notification-related events
- Background processing events

### Lifecycle events

The first two event listeners in the chart belong to lifecycle events<sup>700</sup>. Implementing these event listeners allows you to optionally perform additional tasks when these events run. `install` is triggered as soon as the worker executes, and it's only called once per service worker, allowing you to cache everything you need before the service worker takes control. `activate` fires once a new service worker can control clients and the old service worker is gone. This is a good time to do things such as clearing up old caches used by the previous service worker needed but that are no longer necessary.

Both event listeners have a high adoption: 70.40% of mobile and 70.73% of desktop PWAs implement an `install` event listener and 63.00% of mobile and 64.85% of desktop listen to `activate`. This is expected as the tasks that can be performed inside these events are critical

700. <https://developers.google.com/web/fundamentals/primers/service-workers/lifecycle>

for performance and reliability (for example, precaching<sup>701</sup>). Reasons for not listening to lifecycle events include: using service workers only for notifications (without any caching strategy) or applying caching techniques only to requests made by the site while it is running, a technique called runtime caching<sup>702</sup> which is frequently (but not exclusively) used in combination with precaching techniques.

## Notification-related events

As shown in Figure 16.4 the next group of event listeners in popularity are `push`, `notificationclick` and `notificationclose`, which are related to Web Push Notifications<sup>703</sup>. The most widely adopted is `push`, which lets you listen for push events sent by the server, and it is used by 43.88% of desktop and 45.44% of mobile sites with service workers. This demonstrates how popular web push notifications are in PWAs even when they are not yet available in all browsers<sup>704</sup>.

## Background processing events

The last group of events in Figure 16.4 allow you to run certain tasks in service workers in the background, for example, to synchronize data or retry tasks when the connectivity fails.

Background Sync<sup>705</sup> (via `sync` event listener) allows a web app to delegate a task to the service worker and automatically retry it if it fails or there's no connectivity (in which case the service worker waits for connectivity to be back to automatically retry). Periodic Background Sync<sup>706</sup> (via `periodicSync`) allows running tasks at periodic intervals in the service worker (for example, fetching and caching the top news every morning). Other APIs like Background Fetch<sup>707</sup>, don't show up in the chart, as their usage is still quite low.

As seen, background sync techniques don't have wide adoption yet compared to the others. This is in part because use cases for background sync are less frequent, and the APIs are not yet available across all browsers. Periodic Background Sync<sup>708</sup> also requires the PWA to be installed for it to be used, which makes it unavailable for sites that don't provide "add to home screen"<sup>709</sup> functionality.

Despite that, there are some important reasons for using background sync in modern web apps: one of them being offline analytics (Workbox Analytics uses Background Sync for this<sup>710</sup>), or

701. <https://developers.google.com/web/tools/workbox/modules/workbox-precaching>

702. <https://web.dev/runtime-caching-with-workbox/>

703. <https://developers.google.com/web/fundamentals/push-notifications>

704. <https://caniuse.com/push-api>

705. <https://developers.google.com/web/updates/2015/12/background-sync>

706. <https://web.dev/periodic-background-sync/>

707. <https://developers.google.com/web/updates/2018/12/background-fetch>

708. [https://developer.mozilla.org/docs/Web/API/Web\\_Periodic\\_Background\\_Synchronization\\_API](https://developer.mozilla.org/docs/Web/API/Web_Periodic_Background_Synchronization_API)

709. [https://developer.mozilla.org/docs/Web/Progressive\\_web\\_apps/Add\\_to\\_home\\_screen](https://developer.mozilla.org/docs/Web/Progressive_web_apps/Add_to_home_screen)

710. <https://developers.google.com/web/tools/workbox/modules/workbox-google-analytics>

retrying failed queries due to lack of connectivity (as some search engines do<sup>711</sup>).

**Note:** Unlike previous years, we have decided not to include the `fetch` and `message` events in this analysis, as those can also appear outside service workers, which could lead to a high number of false positives. So, the above analysis is for service worker-specific events. According to 2020 data, `fetch` was used almost as much as `install`.

## Other popular service worker features

Besides event listeners, there are other important service worker functionalities that are interesting to call out, given their usefulness and popularity.

The following two events are quite popular and frequently used in tandem:

- `ServiceWorkerGlobalScope.skipWaiting()`
- `Clients.claim()`

`ServiceWorkerGlobalScope.skipWaiting()` is usually called at the beginning of the `install` event and allows a newly installed service worker to immediately move to the `active` state, even if there's another active service worker. Our analysis showed that it is used in 60.47% of desktop and 59.60% of mobile PWAs.

# 59.60%

Figure 15.5. Percent of mobile sites with service workers that call `skipWaiting()`

`Clients.claim()` is frequently used in combination with `skipWaiting()`, and it allows active service workers to “claim control” of all the clients under its scope. Appears in 48.98% of desktop pages and 47.14% of mobile.

# 47.14%

Figure 15.6. Percent of mobile sites with service workers that call `clients.claim()`

Combining both of the previous events means that a new service worker will immediately come

711. <https://web.dev/google-search-sw/>

into effect, replacing the previous one, without having to wait for active clients (for example, tabs) to be closed and reopen at a later point (for example, a new user session), which is the default behavior. Developers find this technique useful to ensure that every critical update goes through immediately, which explains its wide adoption.

Another interesting aspect to analyze are caching operations, which are frequently used in service workers and are at a core of a PWA experience, since they enable features like offline and help improving performance. The `ServiceWorkerGlobalScope.caches` property returns the `CacheStorage` object<sup>712</sup> associated with a service worker allowing access to the different caches<sup>713</sup>. We've found that it is used in 57.41% desktop and in 57.88% mobile sites that use service workers.

# 57.88%

*Figure 15.7. Percent of mobile sites with service workers that use the service worker cache*

Its high usage is not unexpected as caching allows for reliable and performant web applications, which is often one of the main reasons why developers work on PWAs.

Finally, it's worth taking a look at Navigation Preloads<sup>714</sup>, which allows you to make the requests in parallel with the service worker boot-up time to avoid delaying the requests in those situations. The `NavigationPreloadManager` interface provides a set of methods to implement this technique, and according to our analysis, it is currently used in 11.02% of desktop and 9.78% of mobile sites that use service workers.

# 9.78%

*Figure 15.8. Percent of mobile sites with use navigation preloads*

Navigation Preloads counts with a decent level of adoption, despite the fact that it's not yet available in all browsers<sup>715</sup>. It's a technique that many developers could benefit from, and they can implement it as a progressive enhancement<sup>716</sup>.

712. <https://developer.mozilla.org/docs/Web/API/CacheStorage>

713. <https://developer.mozilla.org/docs/Web/API/Cache>

714. <https://developers.google.com/web/updates/2017/02/navigation-preload>

715. <https://caniuse.com/?search=navigation%20preload%20manager>

716. [https://developer.mozilla.org/docs/Glossary/Progressive\\_Enhancement](https://developer.mozilla.org/docs/Glossary/Progressive_Enhancement)

## Web App Manifests

The Web App Manifest<sup>717</sup> is a JSON file that contains metadata about a web application and it's one of the main components of a PWA, as publishing a web app manifest is one of the preconditions to provide the "add to home screen" functionality, which allows users to install a web app on their device. Other conditions include serving the site via HTTPS, having an icon, and in some browsers (like Chrome and Edge), having a service worker. Take into account that different browsers have different criteria for installation<sup>718</sup>.

Here are some usage stats about Web App Manifests. It's useful to visualize them along with the service worker ones, to start having an idea of the potential percentage of "installable" web applications:

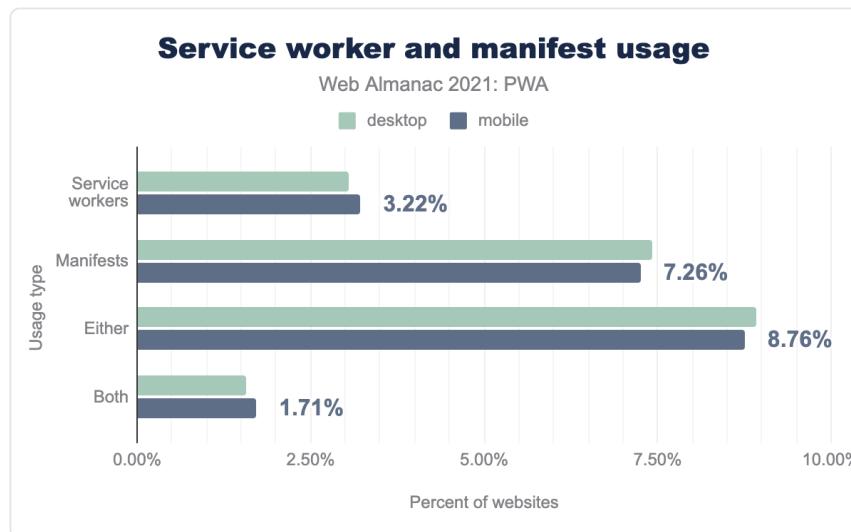


Figure 15.9. Service worker and manifest usage.

Manifests are used on more than twice as many pages as service workers. One of the reasons being that some platforms (like CMSs) automatically generate manifest files for sites, even those without service workers.

On the other hand, service workers can be used without a manifest. For example, some developers might want to add push notifications, caching or offline functionality to their sites, but might not be interested in installability, and therefore, not create a manifest.

717. <https://developer.mozilla.org/docs/Web/Manifest>

718. <https://web.dev/installable-manifest/#in-other-browsers>

In the figure above, we can see that 1.57% of desktop and 1.71% of mobile sites have both a service worker and a manifest. This is a first approximation to the potential percentage of “installable” websites.

Besides having a web app manifest and service worker, the content of the manifest also needs to meet some additional installability criteria<sup>719</sup> for a web application to be installable. We'll analyze each of its properties next.

## Manifest properties

The following chart shows the usage of standard manifest properties<sup>720</sup>, in the group of sites that also have a service worker.

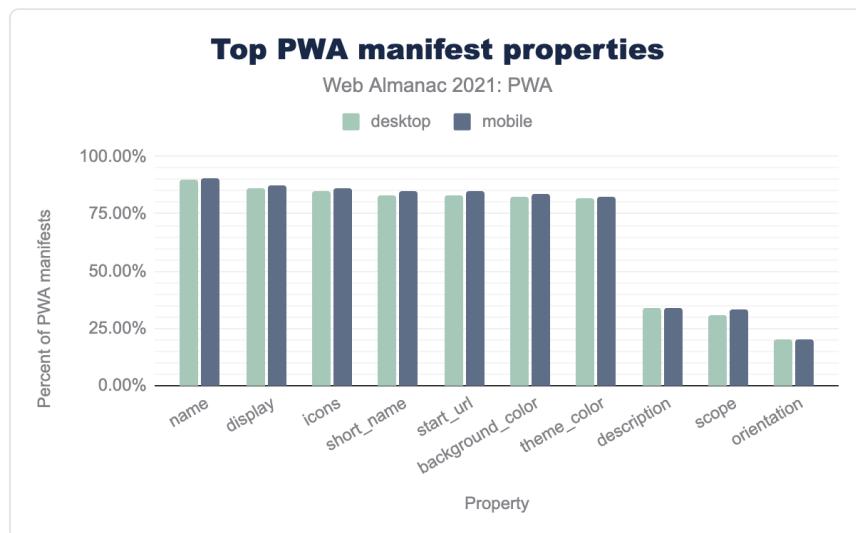


Figure 15.10. Top PWA manifest properties.

This chart is interesting when combined with the Lighthouse Installable Manifests criteria<sup>721</sup>. Lighthouse<sup>722</sup> is a popular tool to analyze the quality of websites and, as we'll see in the Lighthouse Insights section, 61.73% of PWA sites have an installable manifest based on these criteria.

Next we'll analyze each of the Lighthouse installability requirements, one by one, according to the previous chart:

719. <https://web.dev/installable-manifest/>

720. <https://w3c.github.io/manifest/#web-application-manifest>

721. <https://web.dev/installable-manifest/>

722. <https://developers.google.com/web/tools/lighthouse>

- A `name` or `short_name` : The `name` property is present in 90% of sites, while the `short_name` appears on 83.08% and 84.69% of desktop and mobile sites respectively. The high usage of these properties makes sense as both are key attributes: the `name` is displayed in the user's home screen, but if it's too long or the space in the screen is too small, the `short_name` might end up being displayed instead.
- `icon` : This property appears in 84.69% of desktop and 86.11% of mobile sites. Icons are used in various places: the home screen, the OS task switcher, etc. This explains its high adoption.
- `start_url` : This property exists in 82.84% of desktop and 84.66% mobile sites. This is another important property for PWAs, as it indicates what URL will be opened when the user launches the web application.
- `display` : This property is declared in 86.49% of desktop and 87.67% of mobile sites. It's used to indicate the display mode of the website. If it's not indicated, the default value is `browser`, which is the conventional browser tab, so most PWAs declare it to indicate that it should be opened in `standalone` mode instead. The ability to open in standalone mode is one of the things that help create an "app-like" experience.
- `prefer_related_applications` : This property appears in 6.87% of desktop and 7.66% of mobile sites, which seems like a low percentage compared to the rest of the properties in this list. The reason is that Lighthouse doesn't require it to be present, it only suggests against having it set with a value of `true`.

Next, we'll dig deeper into the properties that allow us to define a set of values. To understand which ones are the most widely used.

## Top manifest icon sizes

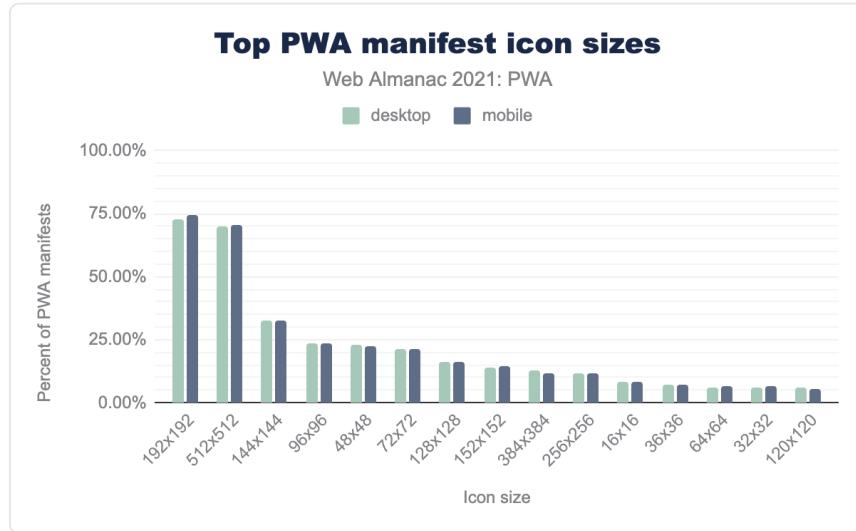


Figure 15.11. Top PWA manifest icon sizes.

The most popular icon sizes, by far, are: 192x192 and 512x512, which are the sizes that Lighthouse recommends<sup>723</sup>. In practice, developers also provide a variety of sizes, to make sure that they look good on various device screens.

723. <https://web.dev/add-manifest/#icons>

## Top manifest display values

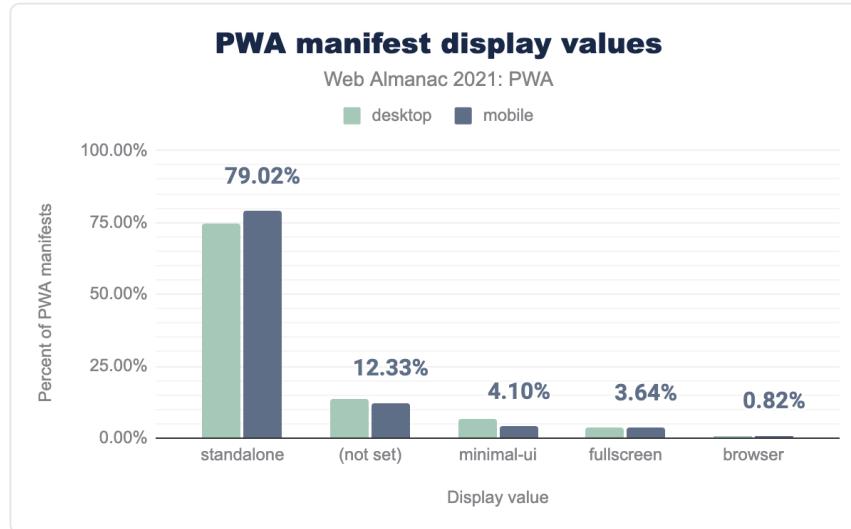


Figure 15.12. PWA manifest display values.

The display property determines the developer's preferred mode for the website. The `standalone` mode makes installed PWAs open without any browser UI element, making it "feel like an app". The chart shows that the most sites with a service worker and manifest uses this value: 74.83% on desktop and 79.02% on mobile.

## Manifests preferring native

Finally, we'll analyze `prefer_related_applications`. If the value of this property is set to `true`, the browser might suggest installing one of the related applications instead of the web app.

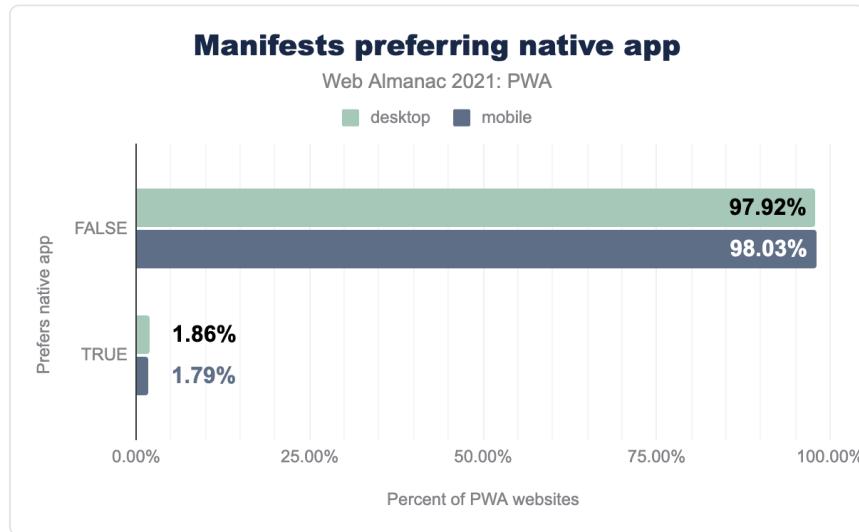


Figure 15.13. Manifests preferring native app.

`prefer_related_applications` appears only in 6.87% of desktop and 7.66% of mobile sites. The chart shows that 97.92% of desktop and 93.03% of mobile sites that defined this property have a value of `false`. This indicates that most PWA developers prefer to offer the PWA experience rather than a native app.

Despite the fact that the vast majority of PWA developers prefer promoting their PWA experiences to native applications, some well-known PWAs (like Twitter), still prefer recommending the native app over the PWA experience. This might be due to a preference of the teams building these experiences, or some specific business needs (lack of some API in the web).

**Note:** Instead of making this decision statically at configuration, developers can also create more dynamic heuristics<sup>724</sup> to promote an experience, for example, based on the user's behavior or other characteristics (device, connection, location, etc.).

## Top manifest categories

In last year's PWA chapter we included a section about manifest categories<sup>725</sup>, showing the percentage of PWAs per industry, based on the manifest categories<sup>726</sup> property.

724. <https://web.dev/define-install-strategy/>

725. <https://almanac.httparchive.org/en/2020/pwa#top-manifest-categories>

726. <https://developer.mozilla.org/docs/Web/Manifest/categories>

This year we decided not to rely on this property to determine how many PWAs of each category are out there, since the usage of this property is incredibly low (less than 1% of sites have this property set).

Given our lack of data on categories and industries using PWAs, we turn to external sources for this information. Mobsted recently published their own analysis of the use of PWAs<sup>727</sup>, which analyzed the percentage of PWAs by industry, among other things:

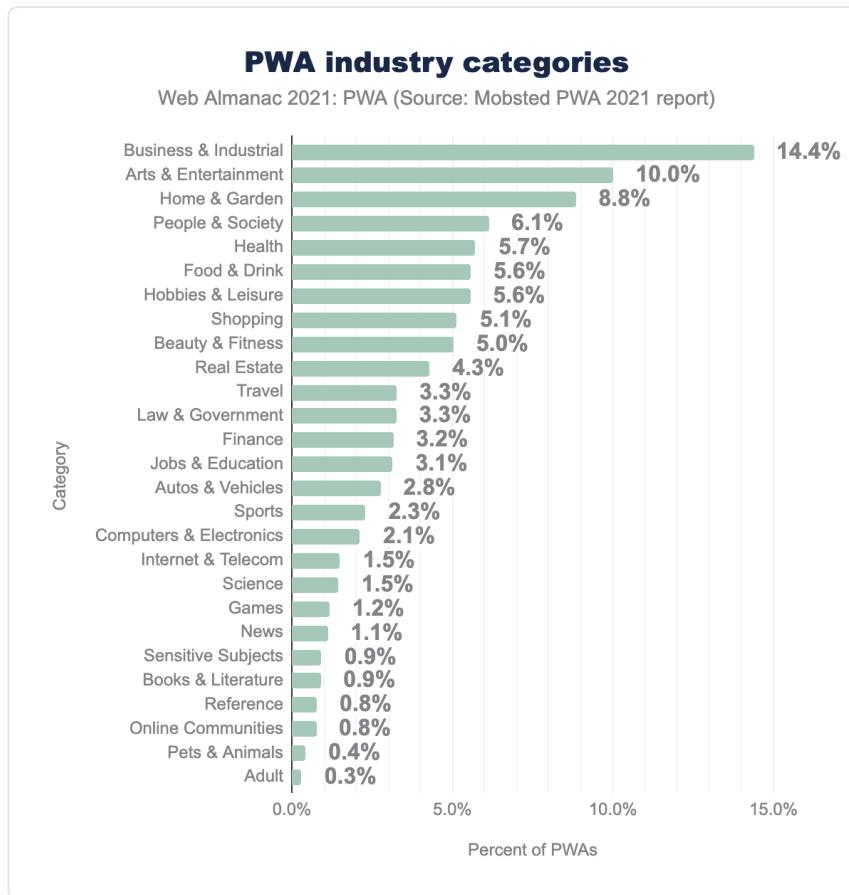


Figure 15.14. PWA industry categories (Source: Mobsted PWA 2021 report<sup>728</sup>).

According to Mobsted's analysis, the most common categories are "Business & Industrial", "Arts & Entertainment", and "Home & Garden". This seems to correlate with last year's analysis of the

727. [https://mobsted.com/world\\_state\\_of\\_pwa\\_2021](https://mobsted.com/world_state_of_pwa_2021)

728. [https://mobsted.com/world\\_state\\_of\\_pwa\\_2021](https://mobsted.com/world_state_of_pwa_2021)

“category” web manifest property<sup>729</sup>, where the top three values were “shopping”, “business” and “entertainment”.

## Lighthouse insights

In the manifest properties section we mentioned the installability requirements<sup>730</sup> that Lighthouse has on web app manifest files. Lighthouse also provides checks for other aspects that make a PWA. It should be noted that the HTTP Archive currently only runs the Lighthouse tests as part of its mobile crawl, as noted in our Methodology.

The following chart shows the percentage of sites that pass each criteria, where “PWA sites” contains stats for sites that have a service worker and a manifest, “All sites” contains data for all the totality sites:

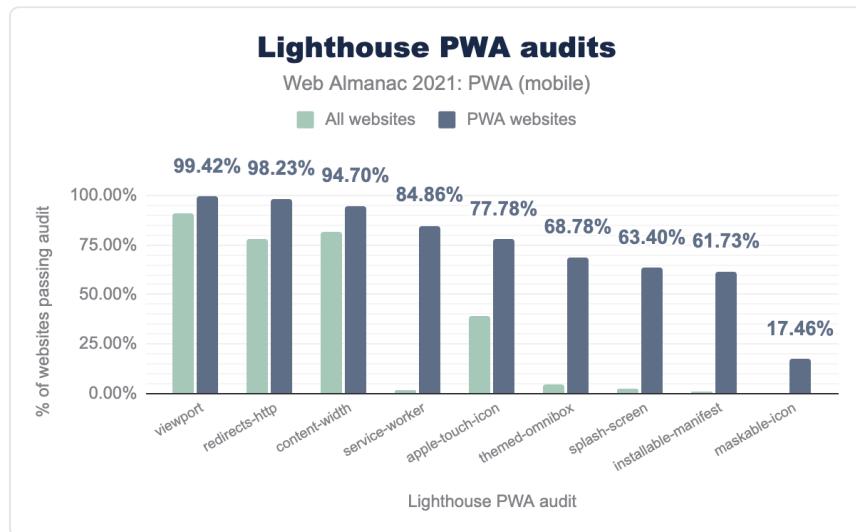


Figure 15.15. Lighthouse PWA audits.

As expected, the table shows that the group of sites that we have identified as PWAs (those having a service worker and manifest) tend to pass each Lighthouse PWA audit. While some audits that are non-PWA specific (for example, setting viewports, or redirecting HTTP to HTTPS) are scored highly by all sites, there is a distinct difference for the PWA-specific audits, with these really only being used by PWA sites.

729. <https://almanac.httparchive.org/en/2020/pwa#top-manifest-categories>

730. <https://web.dev/installable-manifest/>

It's interesting to note that maskable icons<sup>731</sup> have a low pass-rate even for PWA sites compared to the rest of the PWA audits. Using maskable icons lets you enhance the look and feel of icons in Android devices, making them fill up the entire shape assigned to it (like a responsive feature for icons). This feature is optional and mostly interesting for PWAs that offer an installable experience. Unlike other PWA features (like offline), sites that are not PWAs will rarely be interested in it.

Lighthouse also provides a PWA score<sup>732</sup>, based on the "pass rate" of all these audits. The following chart compares the resulting scores among the two groups analyzed before:

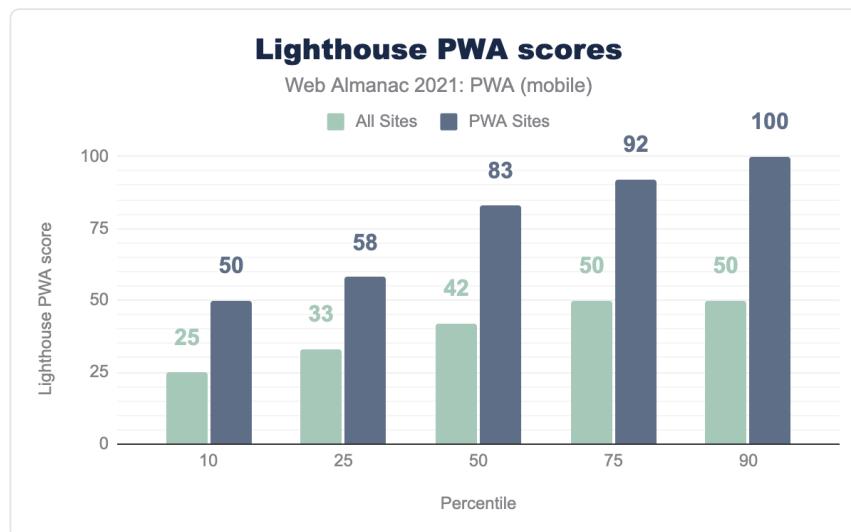


Figure 15.16. Lighthouse PWA scores.

Here are some observations:

- The median score for "PWA sites" is 83, versus 42 for "All sites".
- At the top end we see that for the "PWA sites", at least 10% score the maximum (100) score for PWA. When looking at "All sites" the 75th and 90th percentile reach a value of, at most, 50.
- Taking a look at the lower end of the chart, 90% of "PWA sites" have a Lighthouse PWA score of, at least 50, compared to 25 when we look across all sites.

Once again, the difference between both groups is expected, as "PWA sites" are naturally prone

731. <https://web.dev/maskable-icon/>

732. <https://web.dev/lighthouse-pwa/>

to pass the PWA-specific requirements more often than “All sites”. In any case, the median score of 83 for PWA sites, suggests that a good portion of PWA developers are aligned with best practices.

## Service worker libraries

Service workers can use libraries to take care of common tasks, functionalities and best practices (e.g., to implement caching techniques, push notifications, etc.). The most common way of doing this is by using `importScripts()`<sup>733</sup>, which is the way of importing JavaScript libraries in workers. In other cases, build tools can also inject the code of libraries directly into service workers at build time.

Take into account that not all libraries can be used in worker contexts. Workers don't have access to the `Window`<sup>734</sup>, and therefore, the `Document`<sup>735</sup> object, and have limited access to browser APIs. For that reason, service worker libraries are specifically designed to be used in these contexts.

In this section we'll analyze the popularity of various service worker libraries.

### Popular import scripts

The following chart shows the percentage of usage for the various libraries imported via `importScripts()`.

733. <https://developer.mozilla.org/docs/Web/API/WorkerGlobalScope/importScripts>  
734. <https://developer.mozilla.org/docs/Web/API/Window>  
735. <https://developer.mozilla.org/docs/Web/API/Document>

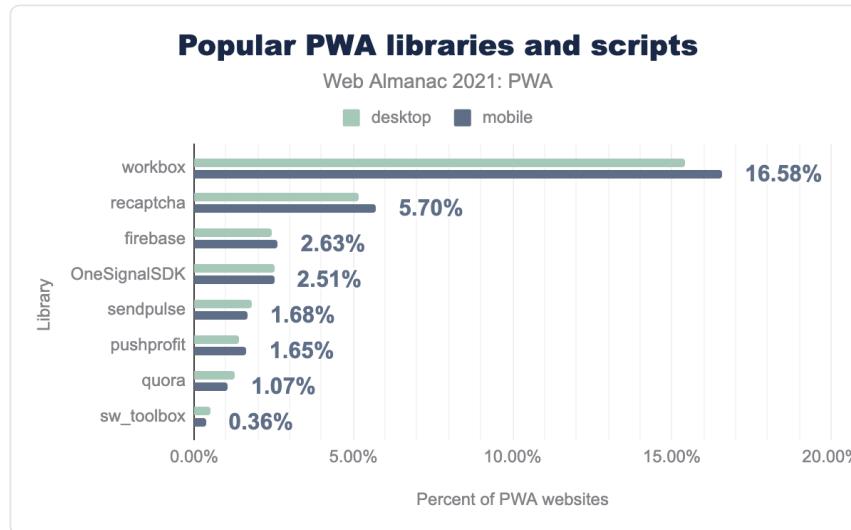


Figure 15.17. Popular PWA libraries and scripts.

Workbox is still the most popular library, being used by 15.43% of desktop and 16.58% of mobile sites with service workers, although this may be interpreted as a proxy for Workbox adoption in general. The next section takes a more holistic and accurate approach to measuring adoption.

It's also important to note that the Workbox predecessor `sw_toolbox`, which had 13.92% of usage in desktop and 12.84% in mobile last year<sup>736</sup> dropped to 0.51% and 0.36% respectively this year. This is in part due to the fact that `sw_toolbox` was deprecated in 2019<sup>737</sup>. It might have taken some time for some popular frameworks and build tools to remove this package, so we are seeing the drop in adoption more clearly this year. Also, our measurement has changed compared to 2020, by adding more sites, which made this metric decrease even more, making it difficult to do a direct year on year comparison.

**Note:** Take into account that `importScripts()` is an API of `WorkerGlobalScope` that can be used in other types of worker context like Web Workers<sup>738</sup>. `reCaptcha`<sup>739</sup>, for example, appears as the second most widely used library, as it uses a web worker that contains an `importScripts()` call to retrieve the reCaptcha JavaScript code. For that reason, we should consider Firebase<sup>740</sup> instead as the second most widely used library in service worker contexts.

736. <https://almanac.httparchive.org/en/2020/pwa#popular-import-scripts>  
 737. <https://github.com/GoogleChromeLabs/sw-toolbox/pull/288>  
 738. [https://developer.mozilla.org/docs/Web/API/Web\\_Workers\\_API/Using\\_web\\_workers](https://developer.mozilla.org/docs/Web/API/Web_Workers_API/Using_web_workers)  
 739. <https://www.google.com/recaptcha/about/>  
 740. <https://firebase.google.com/docs/web/setup>

## Workbox usage

Workbox<sup>741</sup> is a set of libraries that packages a set of common tasks and best practices for building PWAs. According to the previous chart, Workbox is the most popular library in service workers. So, let's take a closer look at how it's used in the wild.

Starting with Workbox 5<sup>742</sup>, the Workbox team has encouraged developers to create custom bundles of the Workbox runtime instead of using `importScripts()` to load `workbox-sw` (the runtime). The Workbox team will continue supporting `workbox-sw`, but the new technique is now the recommended approach. In fact, the defaults for the build tools have switched to prefer that method.

Based on that, we measured sites using any type of Workbox features and found that the number of sites with service workers using it is much higher than noted above: 33.04% of desktop and 32.19% of mobile PWAs.

# 32.19%

Figure 15.18. Percentage of mobile sites with service workers that use the Workbox library.

741. <https://developers.google.com/web/tools/workbox>  
742. <https://github.com/GoogleChrome/workbox/releases/tag/v5.0.0>

## Workbox versions



Figure 15.19. Top 10 workbox versions.

The chart shows that version 6.1.15<sup>743</sup> has the highest level of adoption compared to others. That version was released on April 13th, 2021, and was the latest version at the time of our crawl in July 2021.

There were more versions<sup>744</sup> released since that time, and based on the behavior observed on the chart, we expect them to become the most widely used shortly after being launched.

There are also older versions that still count with wide adoption. The reason for that is that some popular tools have adopted older Workbox versions in the past and continue providing it, namely:

- Version 4.3.1 usage is mostly driven by create-react-app version 3<sup>745</sup>.
- Version 3.0.0 similarly, is included in create-react-app version 2<sup>746</sup>.

743. <https://github.com/GoogleChrome/workbox/releases/tag/v6.1.5>

744. <https://github.com/GoogleChrome/workbox/releases>

745. <https://github.com/facebook/create-react-app/blob/v3.4.4/packages/react-scripts/package.json#L82>

746. <https://github.com/facebook/create-react-app/blob/v2.1.8/packages/react-scripts/package.json#L72>

## Workbox packages

The Workbox library is provided as a set of packages or modules<sup>747</sup> that contain specific functionalities. Each package serves a specific need and can be used together or on its own.

The following table shows the usage of Workbox of the most popular packages:



Figure 15.20. Top workbox packages.

The chart above shows that the following packages are the four most widely used:

- Workbox Core<sup>748</sup>: This package contains the common code that each Workbox module relies on (for example, the code to interact with the console and throw meaningful errors). That's why it's the most widely used.
- Workbox Routing<sup>749</sup>: This package allows to intercept requests and respond to them in different ways. It's also a very common task inside a service worker, so it's quite popular.
- Workbox Precaching<sup>750</sup>: This package allows sites to save some files to the cache while the service worker is installing. This set of files usually constitute the "version" of a PWA (similar to the version of a native app).

747. <https://developers.google.com/web/tools/workbox/modules>

748. <https://developers.google.com/web/tools/workbox/modules/workbox-core>

749. <https://developers.google.com/web/tools/workbox/modules/workbox-routing>

750. <https://developers.google.com/web/tools/workbox/modules/workbox-precaching>

- Workbox Strategies<sup>751</sup>: Unlike precaching, which takes place at the service worker “install” event, this package enables runtime caching strategies to determine how a service worker generates a response after receiving a `fetch` event.

## Workbox strategies

As mentioned, Workbox provides a set of built-in strategies to respond to network requests. The following chart helps us see the adoption of the most popular runtime caching strategies:



Figure 15.21. Top Workbox runtime caching strategies.

`NetworkFirst`, `CacheFirst` and `Stale While Revalidate` are, by far, the most widely used. These strategies let you respond to requests by combining the network and the cache in different ways. For example: the most popular runtime caching strategy: `NetworkFirst` will try to fetch the latest response from the network. If the result is successful, it will put the result in the cache. If the network fails, the cache response will be used.

Other strategies, like `NetworkOnly` and `CacheOnly` will resolve a `fetch()` request by going either to the network or cache, without combining these two options. This might make them less attractive for PWAs, but there are still some use cases where they make sense. For example, they can be combined with plugins<sup>752</sup> to extend their functionality.

751. <https://developers.google.com/web/tools/workbox/modules/workbox-strategies>

752. [https://developers.google.com/web/tools/workbox/modules/workbox-strategies#using\\_plugins](https://developers.google.com/web/tools/workbox/modules/workbox-strategies#using_plugins)

## Web Push notifications

Web Push notifications are one of the most powerful ways of keeping users engaged in a PWA. They can be sent to mobile and desktop users and can be received even when the web app is not in the foreground or even opened (either as a standalone app or in a browser tab).

Here are some usage stats for some most popular notification-related APIs:

Pages subscribe to notifications via the `PushManager` interface of the Push API<sup>753</sup>, which is accessed via the `pushManager` property of the `ServiceWorkerRegistration` interface. It's used by 44.14% of desktop and 45.09% of mobile PWAs.

**45.09%**

*Figure 15.22. Percent of mobile sites with service workers that used some method of the `pushManager` property*

Also as shown in Figure 16.4 related to service worker events, the `push` event listener, which is used to receive push messages, is used by 43.88% of desktop and 45.44% of mobile PWAs.

The service worker interface also allows listening to some events to handle user interactions on notifications. Figure 16.4 shows that `notificationclick` (which captures clicks on notifications) is used by 45.64% of desktop and 46.62% of mobile PWAs.

`notificationclose` is used less frequently: 5.98% of desktop and 6.34% of mobile PWAs. This is expected as there are fewer use cases where it makes sense to listen for the notification "close" event, than for notification "clicks".

**Note:** It's interesting to see that service worker notification events (e.g., `push`, `notificationclick`) have even more usage the `pushManager` property, which is used, for example, to request permission for web push notifications (via `pushManager.subscribe`). One of the reasons for this might be that some sites have implemented web push and decided to roll them back at some point, by eliminating the code to request permission for them, but leaving the service worker code unchanged.

## Web Push notification acceptance rates

For a notification to be useful it has to be timely, precise, and relevant<sup>754</sup>. At the moment of

753. [https://developer.mozilla.org/docs/Web/API/Push\\_API](https://developer.mozilla.org/docs/Web/API/Push_API)

754. <https://developers.google.com/web/fundamentals/push-notifications>

showing the prompt to request permission, the user needs to understand the value of the service. Good notification updates have to provide something useful to the users and related to the reason why the permission was granted.

The following chart comes from the Chrome UX Report and shows the acceptance rates for notifications permission prompts:

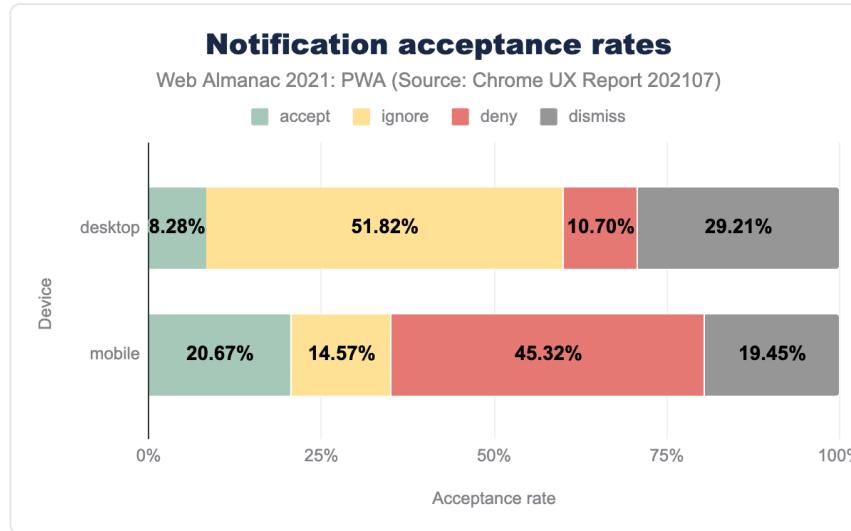


Figure 15.23. Notification acceptance rates.

Mobile has a higher acceptance rate than desktop (20.67% vs 8.28%). This suggests that users tend to find mobile notifications more useful. We can attribute this to two reasons: (1) Users are more familiar with notifications on phones than on desktops, and the utility of a notification in the mobile context is more obvious and (2) the mobile UI for the notification prompt is typically more prominent.

Mobile also has a higher “deny” rate than desktop (45.32% vs 10.70%), and desktop users tend to “ignore” notifications more frequently (19.45% in mobile vs. 29.21 in desktop). The reason for this is that the mobile enrollment UI is much more intrusive than desktop, making the user more frequently decide for either accepting or rejecting the notification. Also, on Desktop devices there are situations when, if a user navigates away from the tab the prompt is dismissed, and the decision is recorded as “ignore” the space to click outside of the prompt to “ignore” the prompt is much bigger.

## Distribution

An important aspect of a PWA is that it allows users to access the web experience in ways beyond typing a URL in the browser URL bar. Users can also install the web app in various ways and access it via a home screen icon. This is one of the most engaging features of native apps, that PWAs also make possible.

Ways to distribute this installable experience include:

- Prompting the user to install the PWA via the add to home screen<sup>755</sup> functionality.
- Uploading the PWA to App Stores by packaging it with Trusted Web Activity (TWA)<sup>756</sup> (currently available in any Android app store, including Google Play and Microsoft Store).

Next, we'll share some stats related to these techniques, to have an idea of the usage and growth of these trends.

### Add to home screen

So far, we have analyzed the pre-conditions for add to home screen, like having a service worker and an installable web app manifest.

In addition to the browser-provided install experience, developers can provide their own custom install flow directly within the app.

The `onbeforeinstallprompt` property of the `Window` object allows the document to capture the event fired when the user is about to be prompted to install a web application. Developers can then decide if they want to show the prompt directly or defer it to show it when they think it's more appropriate.

Our analysis showed that `beforeinstallprompt` is being used in 0.48% of desktop and 0.63% of mobile sites that have a service worker and a manifest.

<sup>755</sup>. [https://developer.mozilla.org/docs/Web/Progressive\\_web\\_apps/Add\\_to\\_home\\_screen](https://developer.mozilla.org/docs/Web/Progressive_web_apps/Add_to_home_screen)

<sup>756</sup>. <https://developer.chrome.com/docs/android/trusted-web-activity/>

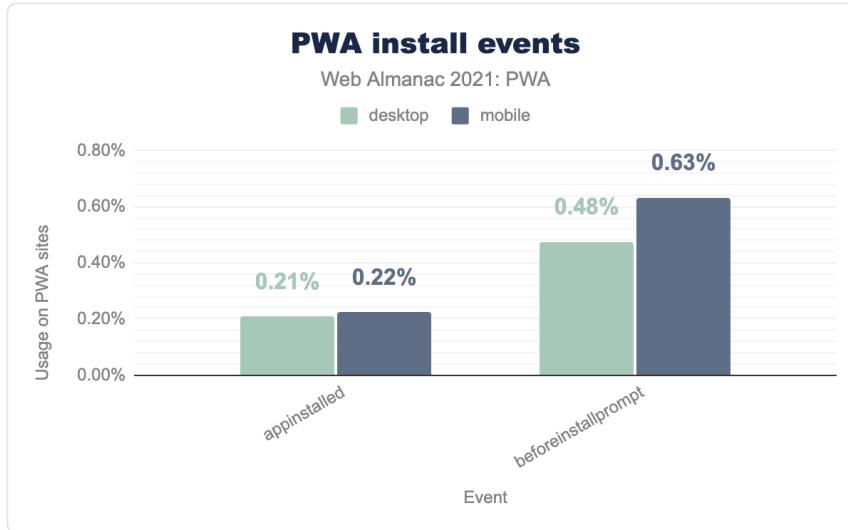


Figure 15.24. PWA install events.

The `BeforeInstallPromptEvent` API is not yet available in all browsers<sup>757</sup>, which explains the relatively low usage. Let's take a look now at the percentage of traffic that this represents:

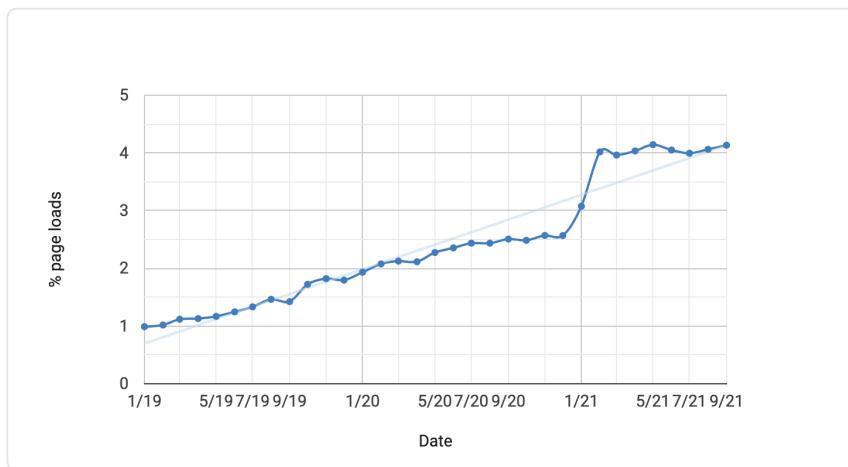


Figure 15.25. Percentage of page view on a page that use `beforeinstallprompt` (Source: Chrome Platform Status<sup>758</sup>)

757. [https://caniuse.com/mdn-api\\_beforeinstallpromptevent](https://caniuse.com/mdn-api_beforeinstallpromptevent)

758. <https://www.chromestatus.com/metrics/feature/timeline/popularity/1436>

According to Chrome Platform Status<sup>759</sup>, the percentage of page loads using this feature is near 4%<sup>760</sup>, which suggests that some high traffic sites might be using it. Additionally, we can see that there was a 2.5 percentage point growth in adoption compared to last year.

## App Store distribution

Historically, developers have built web-based mobile applications and uploaded them to App Stores as an alternative to building apps with OS-specific languages (Java or Kotlin for Android, Objective-C or Swift for iOS). The most common approach is to use a cross-platform, hybrid solution like Cordova<sup>761</sup> that allows one to write the code once and generate multiple versions of it for various platforms. The resulting code usually uses the WebView<sup>762</sup> to render web content, but also provides a series of non-standard APIs that can access features from the device.

WebView-based apps may look similar to native apps, but certainly there are some caveats. Since a WebView is just a rendering engine, users may have different experiences than in a full browser. The latest browser APIs might not be available and most importantly, cookies are not shareable between WebViews and browsers.

TWAs allow you to package your PWA into a native application shell and upload it to some App Stores. Unlike WebView-based solutions, a TWA is not just a rendering engine; it's the full browser running in fullscreen mode. For that reason, it's feature-complete and evergreen, meaning that it's always up to date and will give you access to the latest web APIs.

Developers can package their PWAs into native apps with TWA directly, by using Android Studio<sup>763</sup>, but there are several tools that make this task much easier. Next, we'll analyze two of them: PWA Builder and Bubblewrap.

## PWA Builder

PWA Builder<sup>764</sup> is an open-source project that can help web developers to build Progressive Web Apps and package them for app stores like the Microsoft Store and Google Play Store. It starts by reviewing a provided URL to check for an available manifest, service worker, and SSL.

PWA Builder reviewed 200k URLs over a 3-month timeslot<sup>765</sup> and discovered that:

- 75% had a manifest detected

759. <https://www.chromestatus.com/metrics/feature/timeline/popularity/1436>

760. <https://www.chromestatus.com/metrics/feature/timeline/popularity/1436>

761. <https://cordova.apache.org/>

762. <https://developer.android.com/reference/android/webkit/WebView>

763. <https://developer.chrome.com/docs/android/trusted-web-activity/integration-guide/>

764. <https://www.pwabuilder.com/>

765. <https://twitter.com/pwabuilder/status/1454250060326318082?s=21>

- 11.5% had a service worker detected
- 9.6% are installable PWAs from the browser (manifest and SW and https)

## Bubblewrap

Bubblewrap<sup>766</sup> is a set of tools and libraries designed to help developers to create, build, and update projects for Android apps that launch PWAs using TWA.

By using Bubblewrap, developers don't need to be aware of any details around Android tools (like Android Studio), which makes it very easy to use for web developers.

While we don't have usage stats for Bubblewrap, there are some notable tools that are known to rely on it. For example, PWA Builder and PWA2APK<sup>767</sup> are powered by Bubblewrap.

## Conclusion

Six years after the term "Progressive Web Apps" was coined, the adoption of its core technologies continues to grow. Service workers will soon control 20% of web traffic, and sites continue adding more capabilities each year.

In 2021, developers have a diverse range of options to build and distribute their web applications, including tools that allow them to take on the most common tasks, and offer easy ways of uploading these experiences to app stores.

Year over year the web continues demonstrating that applications that used to be built only with OS-specific languages can be developed with web technologies and companies continue investing<sup>768</sup> in bringing these app-like experiences to the web.

We hope this analysis will assist you in making more informed decisions around your PWA projects. We are looking forward to seeing how much all these trends will grow in 2022!

---

766. <https://github.com/GoogleChromeLabs/bubblewrap>

767. <https://appmaker.xyz/pwa-to-apk>

768. <https://www.theverge.com/2021/10/26/22738125/adobe-photoshop-illustrator-web-announced>

## Author

---



### Demian Renzulli

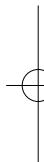
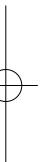
 @drenzulli  demianrenzulli

Demian is a member of Google's Web Ecosystems Consulting team, born in Buenos Aires, Argentina and currently based in New York. His focus is on Progressive Web Apps and Advanced Capabilities. He often writes at [web.dev](https://web.dev)<sup>769</sup>.

---

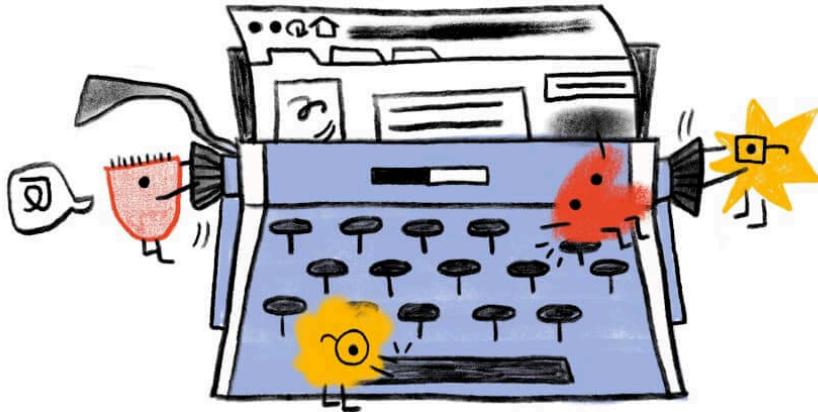
---

<sup>769</sup>. <https://web.dev/authors/demianrenzulli/>



# Part III Chapter 16

# CMS



**Written by Alon Kochba**

*Reviewed by Alan Kent, Andrey Lipattsev, Chris Sater, and John Teague*

*Analyzed by Rick Viscomi and Tosin Arasi*

*Edited by Shaina Hantsis*

## Introduction

In this chapter, we seek to help understand the current state of the CMS ecosystems and the growing role they play in shaping users' perception of how content can be consumed and experienced on the web. Our goal is to discuss aspects related to the CMS landscape in general, and the characteristics of web pages generated by these systems.

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 platform space and bring to it our curiosity along with deep expertise on some of the major players in the space.

These platforms play a key role for us to succeed in our collective quest for a fast and resilient web. This has become increasingly apparent in the past year, and we expect it to continue to be the case going forward.

It is important to take some of these comparisons with a grain of salt, considering the variability between CMSs, and the differing types of user content which are built on these platforms.

In some of the sections, we focus only on the top CMSs in terms of adoption, due to the large number of CMS platforms.

**TLDR:** We discover that almost half of all the sites in the world are created using a CMS. While the top 10 most popular CMS list remains relatively stable year-over-year, there are some interesting changes in market share. The performance of CMS-built sites has improved dramatically since the last time we checked.

Let's dive into our analysis.

*Disclaimer: Alon works at Wix where he leads the web performance efforts, but opinions are his own.*

## What is a CMS?

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 web.

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. CMSs also provide administrative capabilities aimed at making it easy for users to upload and manage content as needed.

There is great variability in 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.

Our definition of a CMS in this chapter uses Wappalyzer's definition<sup>770</sup> of a CMS.

We encourage CMSs to contribute to this open-source project<sup>771</sup> to improve detection and

---

770. <https://www.wappalyzer.com/technologies/cms>

771. <https://github.com/AlisaIO/wappalyzer>

classification in the future.

Shopify, Magento, Webflow, and some other platforms do not appear in this chapter's analysis, because they are not marked as a CMS in Wappalyzer.

Ecommerce platforms make a substantial part of non-CMS sites and are covered in the Ecommerce chapter. For example, Shopify grew substantially in the past year and accounted for 3.7% of websites in July according to W3Techs<sup>772</sup>.

Our research identified over 200 individual CMSs, with these ranging from a single install to millions on a single CMS.

Some of them are open source (e.g., WordPress and Joomla) and some of them are proprietary (e.g., Wix and Squarespace). 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.

## 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.

<sup>772</sup>. [https://w3techs.com/technologies/history\\_overview/content\\_management/all/q](https://w3techs.com/technologies/history_overview/content_management/all/q)



Figure 16.1. CMS adoption year-over-year.

As of July 2021, over 45% of public websites are powered by a CMS platform, indicating growth of over 7% from 2020<sup>773</sup>. This breaks down to 45% on desktop, up from 42% in 2019, and 46% on mobile, up from 42% in 2020.

It is interesting to compare these numbers with another commonly used dataset, such as W3Techs<sup>774</sup>, which reported that as of July 2021, 64.6% of websites are created using a CMS, up from 59.2% in July 2020, which is an increase of over 9%.

The deviation between our analysis and W3Techs' analysis can be explained by a difference in research methodologies, and the definition of what is a CMS.

W3Techs definition is the following: “Content Management Systems are applications for creating and managing the content of a website. We include all such systems in this category, also systems that are often classified as wikis, blog engines, discussion boards, static site generators, website editors or any type of software that provides website content.”

As mentioned previously, Wappalyzer has a stricter definition of a CMS, which excludes some major CMSs which appear in W3Techs reports.

You can read more about ours on the Methodology page.

<sup>773</sup>. <https://almanac.httparchive.org/en/2020/cms#cms-adoption>

<sup>774</sup>. [https://w3techs.com/technologies/history\\_overview/content\\_management/all/q](https://w3techs.com/technologies/history_overview/content_management/all/q)

## CMS adoption by geography

CMS platforms are extensively used around the world, with some variance by country.



Figure 16.2. CMS adoption by country.

Among the geographies with the highest number of websites, CMS adoption percentage is the highest in the US, Italy, and Spain, where 46%–47% of mobile sites visited by users are built with a CMS. India and Brazil have the lowest adoption with only 35% and 37%.

We can also split this data into subregions<sup>775</sup> around the globe, sorted by the most popular regions, to better identify macro-trends:

775. <https://github.com/GoogleChrome/CrUX/blob/main/utils/countries.json>

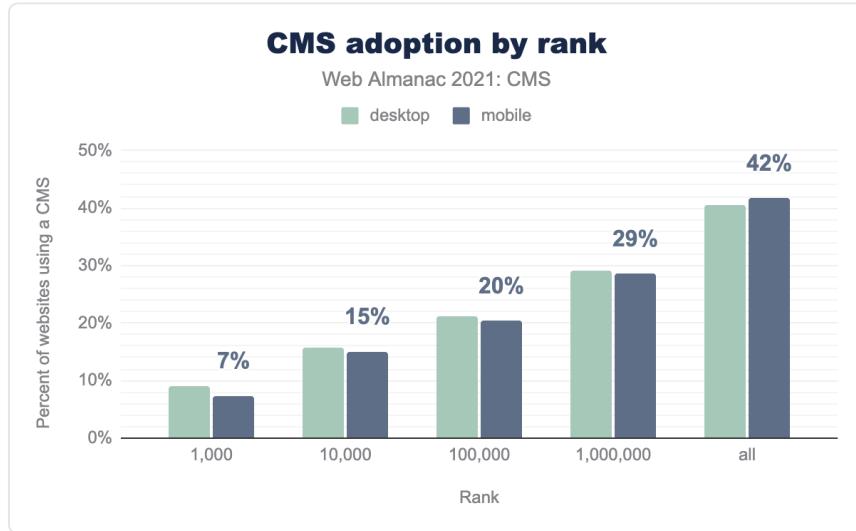


Figure 16.3. CMS adoption by subregion.

Adoption is highest in Southern Europe where half of the sites are using a CMS, and lowest in Eastern Asia where only a third of sites in our dataset use a CMS.

### CMS adoption by rank

We also examined CMS adoption by the estimated rank of the sites.



*Figure 16.4. CMS adoption by rank.*

CMSs account for only 7% of the top 1,000 mobile websites, compared to 42% of the complete dataset of all sites in our analysis. This can be explained by the fact that smaller businesses and websites tend to use a CMS due to the ease of use, and the higher ranked websites tend to be built with proprietary solutions by professional web developers. With the continuing growth in usage of CMS platforms, it would be interesting to see if CMS platforms will also be able to increase adoption rates among the higher-ranking sites in the coming years.

## Top CMSs



Figure 16.5. CMS adoption share.

Among all websites that use a CMS, WordPress sites account for a large part of the relative market share, with over 75% adoption, followed by Joomla, Drupal, Wix, and Squarespace.



Figure 16.6. Top 5 CMSs year-over-year.

Drilling into the adoption by CMS across all websites, out of 218 different CMS platforms only 5 platforms had over 1% of usage.

WordPress, the most commonly used platform, is used by 33.6% of these websites, up from 31.4% in 2020, a 7% increase in total adoption.

In percentage terms, Joomla and Drupal adoption is dropping—Joomla sites accounted for 1.9% of websites, down from 2.1% last year (9.5% decrease), and Drupal dropped from 2% to 1.8% (10% decrease). Absolute adoption did increase in terms of number of sites measured, but as a percentage of both overall CMS usage and of our (ever increasing!) data set, it is smaller.

Wix adoption grew from 1.2% to 1.6% (33% increase) and Squarespace grew from 0.9% to 1% (11% increase).

Examining the adoption of these sites built on CMS platforms by their rank magnitude<sup>776</sup> reveals an interesting distribution between platforms.

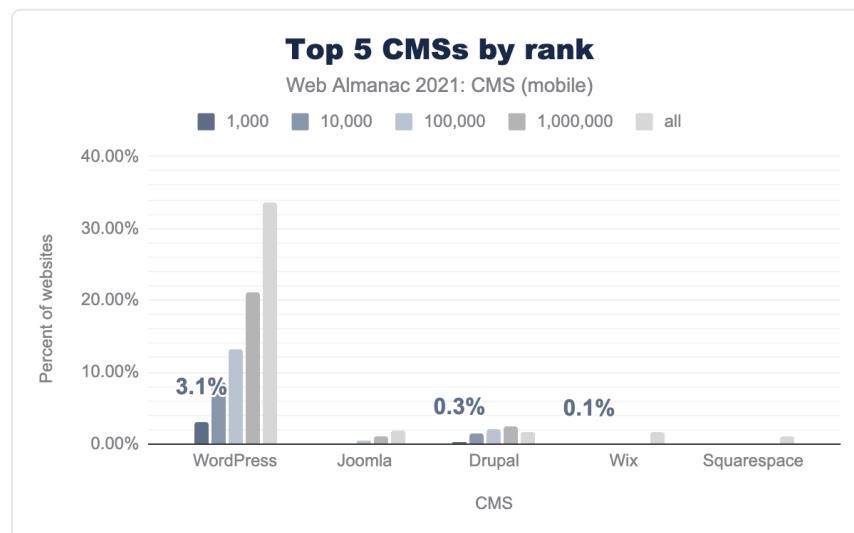


Figure 16.7. Top 5 CMSs by rank.

3.1% of mobile sites in the top 1K are built with WordPress, and 33.6% of all sites. Drupal maintains a higher adoption rate within the mid-ranged rankings (10K–1M), while most of Wix and Squarespace sites are ranked outside the top 1M sites.

<sup>776</sup> <https://developers.google.com/web/updates/2021/03/crxus-rank-magnitude>

## CMS user experience

An important aspect of CMSs is the user experience they provide, for users visiting sites built on these platforms. We attempt to examine these experiences through Real User Measurements (RUM), provided by the Chrome User Experience Report<sup>778</sup> (CrUX), and synthetic testing using Lighthouse.

### Core Web Vitals

2021 was a great year for web performance, with a growing focus on Core Web Vitals<sup>779</sup>, which helped nudge many platforms in the right direction to focus on improving their user experience and loading times. More importantly, it provides users with the right tools and guidance to monitor and improve their website performance. As a result, we saw large performance improvements from many platforms, which continue to evolve, gradually making user experience better across the web, which is a big win for all of us.

The Core Web Vitals Technology Report<sup>779</sup> can be used to drill into this data and view the progress of each technology updated on a monthly basis.

In this section we focused on data from July 2021 to provide a consistent timeframe for data presented across the Web Almanac, and examined 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:

- 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. The Performance chapter covers these in more detail, but here we are interested in looking at these metrics specifically in terms of CMSs.

Initially, let's review the 10 CMS platforms with the highest number of origins, and examine what percentage of sites on each platform have a *passing* grade, meaning that the 75th percentile of each of the above metrics must be in the “good” (green) range for each site.

777. <https://developers.google.com/web/tools/chrome-user-experience-report>

778. <https://web.dev/vitals/#core-web-vitals>

779. <https://httparchive.org/reports/cwv-tech>



*Figure 16.8. Top 10 CMSs core web vitals performance.*

We can see that desktop visitors generally score slightly better than mobile, which can be explained by weaker mobile devices and poorer connections.

The large difference between mobile and desktop in certain platforms also suggests considerably different pages that are served to users on different devices.

In July, for mobile devices, TYPO3 CMS (used mostly in European countries) had the largest percentage of passing sites, with 46% of mobile sites passing all three CWVs. WordPress, Squarespace, and Adobe Experience Manager had less than 20% of their sites pass.

Desktop device experience was slightly better, with 1C-Bitrix (used mostly in Russia) having the largest percentage of 56% sites passing CWVs. WordPress had the lowest ratio of passing sites, with only 26%.

*Duda deserves an honorable mention, with 47% sites passing in August and overall great progress since last year. They were not included in this report due to broken data collection in July, related to a*

wrong detection in Wappalyzer<sup>780</sup>, incorrectly inflating their origins, and reducing their CWV percentage.

We can also evaluate the progress of these CMS platforms compared to last year's data, focusing on mobile views:



Figure 16.9. Top 10 CMSs core web vitals performance for mobile views year-over-year.

All of these CMSs showed an improvement in the percentage of origins with good CWVs since August 2020. Wix and Squarespace made the most noticeable progress, closing the gap from the other CMSs.

Let's drill into the three Core Web Vitals, to see where each platform has room to improve, and which metrics improved the most since last year:

<sup>780</sup> <https://github.com/AliaislO/wappalyzer/pull/4189>

## Largest Contentful Paint (LCP)

Largest Contentful Paint (LCP) measures the point in time 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.

A “good” LCP is regarded as being under 2.5 seconds.

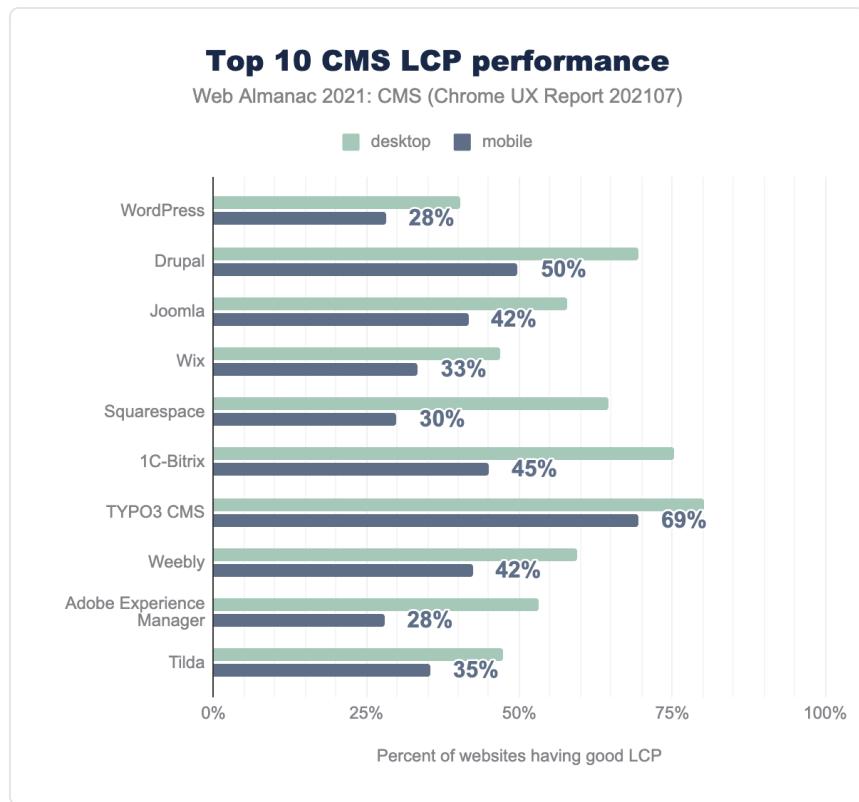


Figure 16.10. Top 10 CMSs LCP performance.

TYPO3 CMS had the best LCP scores with 69% of origins having a “good” LCP experience, while WordPress and Adobe Experience Manager have the worst LCP scores, with only 28% of origins having a good LCP score.

In general, it seems that most platforms are struggling with the LCP metric. This probably relates to the fact that the LCP is dependent on the download of image/font/CSS and then displaying the appropriate HTML elements. Achieving this in under 2.5 seconds for all device

types and connection speeds can be challenging. Improving LCP scores usually involves the correct use of caching, pre-loading, resource prioritization, and lazy loading of other competing resources.



Figure 16.11. Top 10 CMSs LCP performance for mobile views year-over-year.

We can see that all CMSs improved their LCP in the past year, but most of them had modest improvements. The largest jump came from Wix and Squarespace, who had very low LCP scores last year. Tilda also seems to have made considerable progress.

### First Input Delay (FID)

First Input Delay (FID) measures the time from when a user first interacts with the page (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 able to process that interaction. A “fast” FID from a user’s perspective would be almost 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.

A “good” FID is regarded as being under 100 milliseconds.

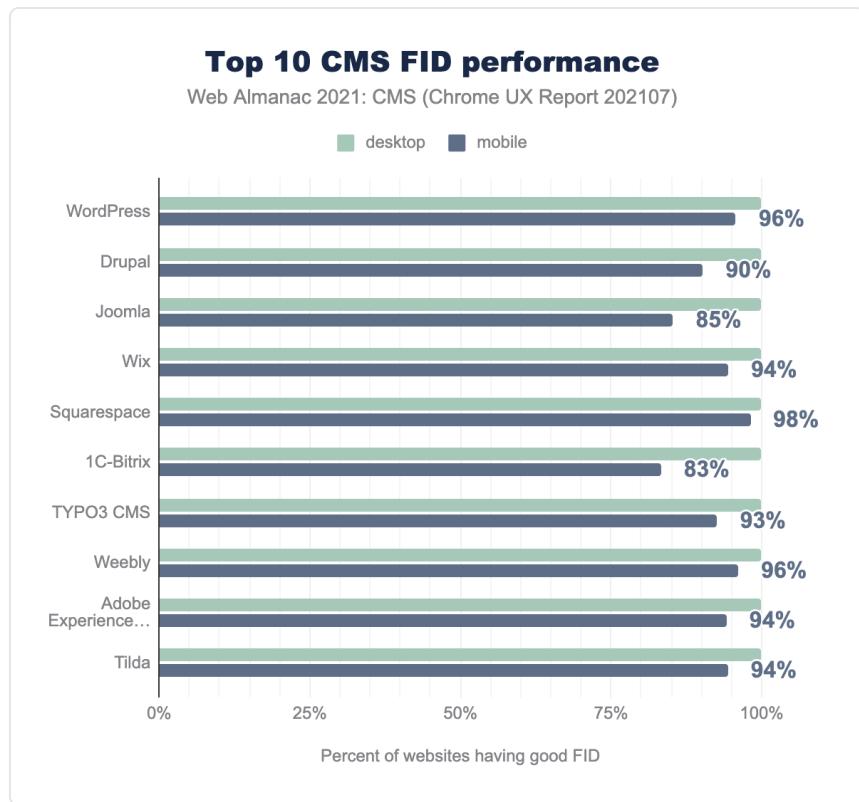


Figure 16.12. Top 10 CMSs FID performance.

FID is very good for most CMSs on desktop, with all platforms scoring a perfect 100%. Most CMSs also deliver a good mobile FID of over 90%, except Bitrix and Joomla with only 83% and 85% of origins having a good FID.

The fact that almost all platforms manage to deliver a good FID, has recently raised questions about the strictness of this metric. The Chrome team recently published an article<sup>781</sup>, which detailed the thoughts towards having a better responsiveness metric in the future.

<sup>781</sup>. <https://web.dev/responsiveness/>



*Figure 16.13. Top 10 CMSs FID performance for mobile views year-over-year.*

Yearly data shows that all these CMSs managed to improve their FID over the past year. Wix had the most catching up to do on FID, and considerably improved their numbers. Joomla and Bitrix had the lowest FID scores this year, but still managed to improve.

### Cumulative Layout Shift (CLS)

Cumulative Layout Shift (CLS) measures the visual stability of content on a web page, measuring the largest burst of layout shift scores for every unexpected layout shift that occurs during the entire lifespan of a page that was not caused by direct user interactions.

A layout shift occurs any time a visible element changes its position from one rendered frame to the next.

The CLS metric has evolved<sup>782</sup> in the past year, mainly introducing the concept of Session Windows, to be fairer to long-lived pages and Single Page Apps (SPAs).

A score of 0.1 or below is measured as “good”, over 0.25 as “poor”, and anything in between as “needs improvement”.



Figure 16.14. Top 10 CMSs CLS performance.

Wix had the best CLS score, with 81% of mobile origins having a “good” CLS. Adobe Experience Manager had the lowest CLS scores, with only 44% of mobile origins having a good CLS. Because layout shifts can usually be avoided, regardless of connection speeds—all platforms should strive to improve these numbers by reducing layout shifts<sup>783</sup> to the bare minimum.

782. <https://web.dev/evolving-cls/>

783. <https://web.dev/optimize-cls/>

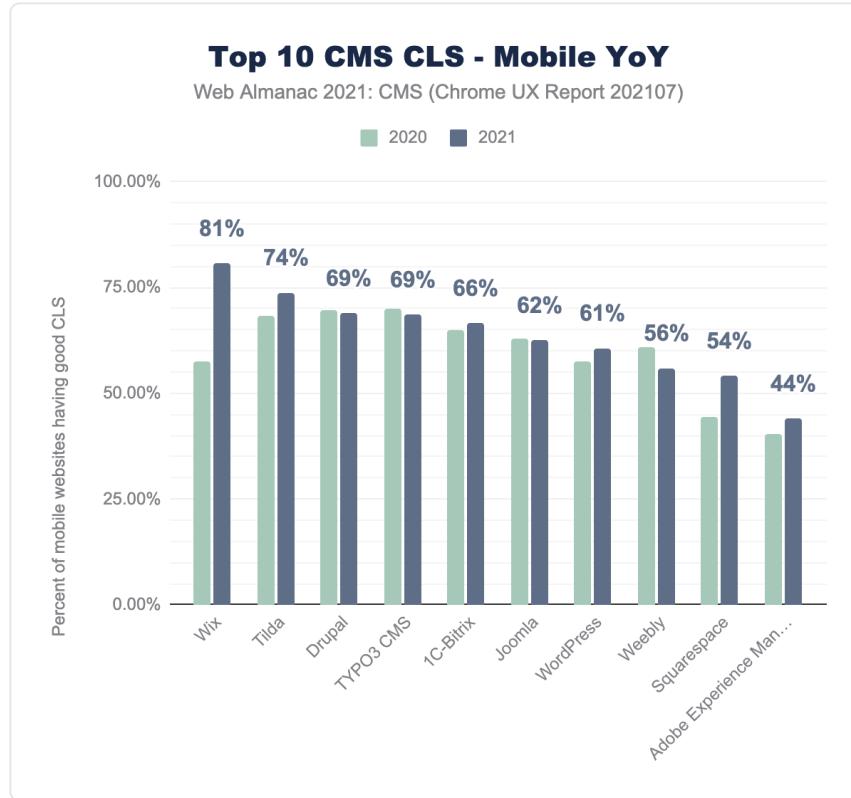


Figure 16.15. Top 10 CMSs CLS performance for mobile views year-over-year.

Comparing yearly data, we can see that most CMSs made some progress, or benefited from the change to a windowed CLS metric. However, we can see that certain CMSs such as Weebly regressed in CLS scores over the past year.

## Lighthouse

Lighthouse<sup>784</sup> is an open-source, automated tool for improving the quality of web pages. 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, best practices, and more. Lighthouse reports provide lab data, a way developers can get suggestions on how to improve website performance, but the Lighthouse score has no direct implications on the actual field data collected by CrUX<sup>785</sup>. You can read more on Lighthouse and the correlation between its lab scores and field data<sup>786</sup>.

784. <https://developers.google.com/web/tools/lighthouse/>

785. <https://developers.google.com/web/tools/chrome-user-experience-report>

786. <https://web.dev/lab-and-field-data-differences/>

HTTP Archive runs Lighthouse on all its mobile web pages (unfortunately, no desktop results), which are also throttled to emulate a slow 4G connection with a CPU slowdown.

We can analyze this data to provide another perspective on CMS performance, using the results of these synthetic tests, which also include metrics that are not tracked in CrUX.

## Performance score

The Lighthouse performance score<sup>787</sup> is a weighted average of several metric scores.

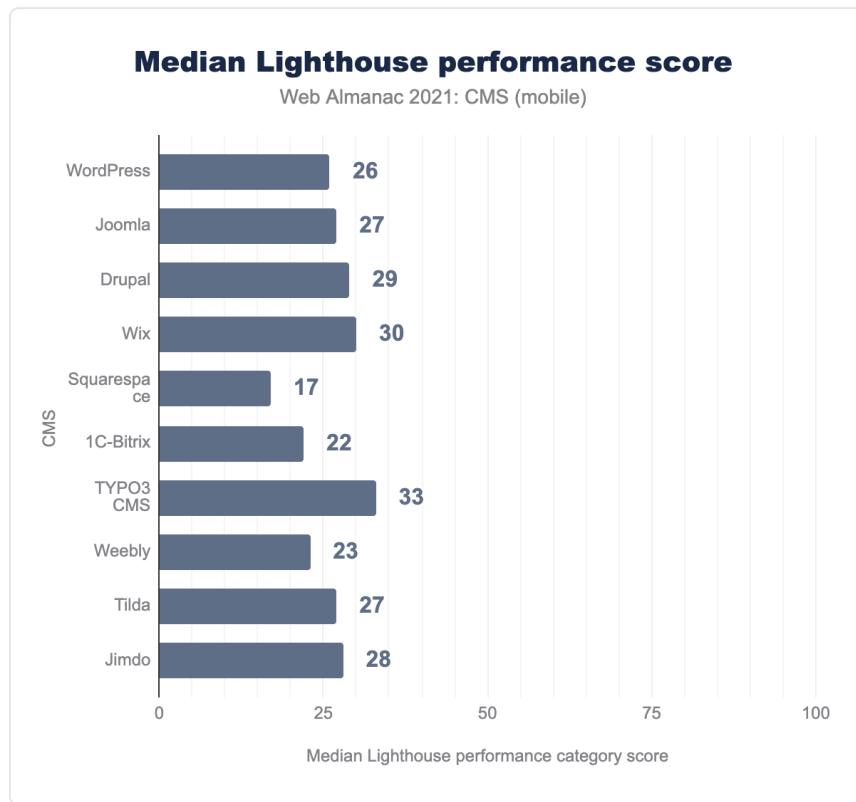


Figure 16.16. Top 10 CMSs median Lighthouse performance score.

We can see that the median performance scores for all the top platforms on mobile are low, ranging from 17 to 33. As we saw above, this does not directly imply bad results<sup>788</sup> in mobile field data but does imply that all platforms have room for improvements, especially for low-end

<sup>787</sup> <https://web.dev/performance-scoring/>

<sup>788</sup> <https://philipwalton.com/articles/my-challenge-to-the-web-performance-community/>

devices and network connections similar to those Lighthouse attempts to emulate.

### SEO score

Search Engine Optimization (or SEO) is the practice of improving a website to make it more easily found in search engines. This is covered more in-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 a site appropriately in search engine results. Compared to a custom-created website, one might expect a CMS to provide good SEO capabilities, and the Lighthouse scores in this category are appropriately high.

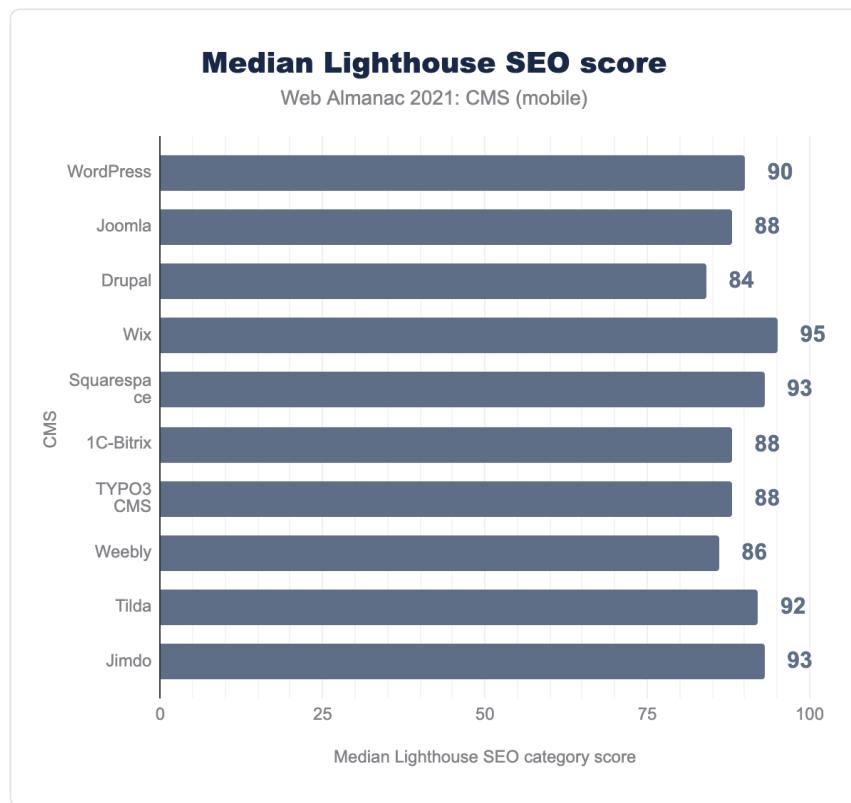


Figure 16.17. Top 10 CMSs median Lighthouse SEO score.

The median SEO score in all of the top 10 platforms is over 84, with Drupal scoring the lowest and Wix scoring the highest with a median score of 95.

## Accessibility score

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. Read more in our Accessibility chapter.

Lighthouse provides a set of accessibility audits, and it returns a weighted average of all of them (see Scoring Details<sup>789</sup> 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 16.18. Top 10 CMSs median Lighthouse accessibility score.

789. <https://web.dev/accessibility-scoring/>

The median Lighthouse accessibility score for the top 10 CMSs ranges between 76 and 91. Squarespace and Weebly have the highest scores of 91, while Tilda had the lowest accessibility scores.

## Best practices

The Lighthouse best practices<sup>790</sup> try to ensure that web pages are following best practices for the web, for a variety of different metrics, such as supporting HTTPS, no errors logged in the console, and more.



Figure 16.19. Top 10 CMSs median Lighthouse best practices score.

Wix had the highest median best practices score of 93, while many of the other top 10 platforms share the lowest score of 73.

<sup>790</sup>. <https://web.dev/lighthouse-best-practices/>

## Resource weights

We can also use HTTP Archive data to analyze the weight of resources used across different platforms, to highlight possible opportunities. Page loading performance does not exclusively depend on the number of downloaded bytes, but fewer bytes necessary to load a page results in reduced costs, carbon emissions, and potentially faster performance, especially for slower connections.

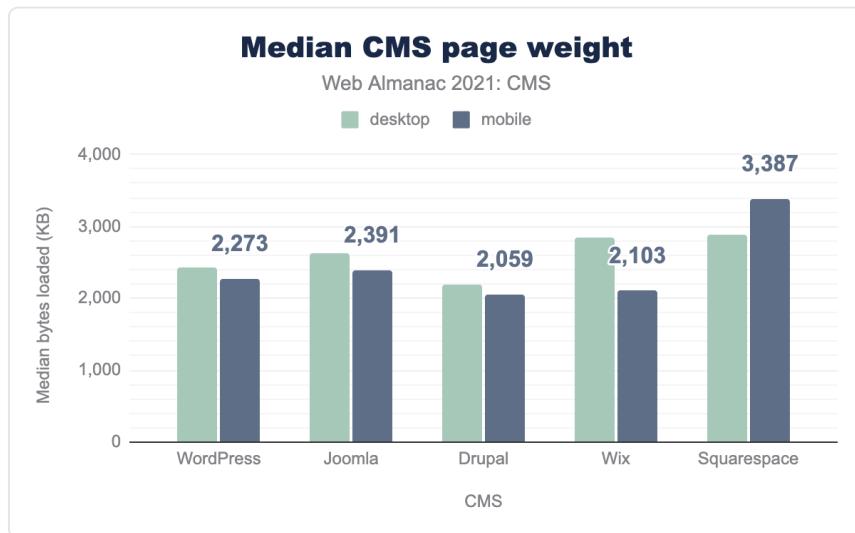


Figure 16.20. Top 5 CMSs median page weight.

Most of the top 5 CMSs deliver a median page weight of around ~2 MB, except Squarespace which delivers a larger ~3.3 MB. Squarespace is the only platform that delivers more bytes in mobile views than on desktop.



Figure 16.21. Top 5 CMSs median page weight.

The distribution of page weight in each platform's percentiles is substantial, probably related to the difference in user content across different web pages, the number of images used, plugins, etc. The smallest pages delivered per platform come from Drupal, which only sends 595 KB for their 10th percentile of visits. The largest pages come from Squarespace, with ~9.6 MB delivered for their 90th percentile of visits.

## Page Weight Breakdown

Page Weight is a sum of resources used. We can attempt to evaluate these different resource sizes across different CMSs.

### Images

Images, which are usually the heaviest resource, account for a large portion of the resource weight.



Figure 16.22. Top 5 CMSs median image weight.

Wix delivers substantially fewer image bytes, with only 357 KB delivered on the median of mobile views, suggesting good use of image compression and lazy image loading. All of the other top 5 platforms deliver over 1 MB of images, with Squarespace delivering the largest ~1.7 MB.

Advanced image formats provide a considerable improvement in compression, enabling resource savings and faster site loading. WebP is commonly supported in all major browsers today, with over 95% support<sup>791</sup>. In addition, there are several newer image formats gaining popularity and adoption, namely AVIF<sup>792</sup>, and JPEG-XL<sup>793</sup> which is still not complete but has outstanding potential.

We can examine the usage of the different image formats across the top CMSs:

791. <https://caniuse.com/webp>

792. <https://caniuse.com/avif>

793. <https://jpegxl.info/>

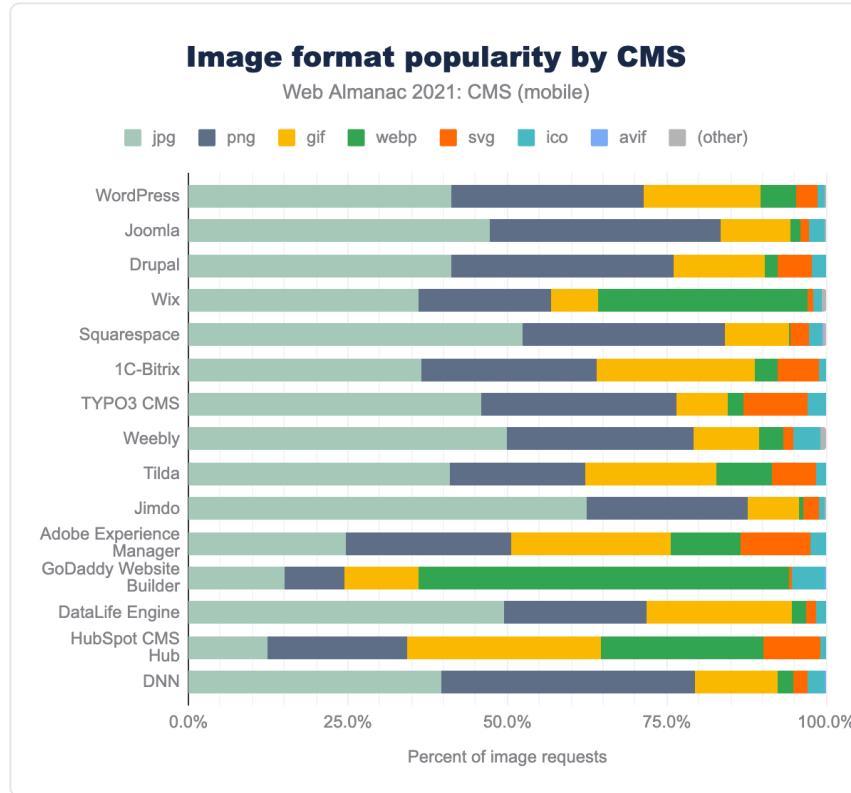


Figure 16.23. Top 15 CMSs image format popularity.

GoDaddy Website Builder and Wix make the most use of WebP, with ~58% and 33% adoption respectively, while WordPress, Joomla, and Drupal barely serve WebP—only ~5.7% of images served by WordPress sites are WebP. AVIF is barely used by these platforms, with less than ~0.1% on all platforms.

With the growing support of WebP<sup>794</sup>, it seems all platforms have work to do to reduce the usage of the older JPEG and PNG formats, where it is applicable without compromising on image quality.

<sup>794</sup>. <https://caniuse.com/webp>

## JavaScript



Figure 16.24. Top 5 CMSs median JavaScript weight.

The largest five CMSs all deliver pages that rely on JavaScript, with Drupal delivering the least amount of JavaScript bytes—372 KB on mobile, while Wix delivers the most JavaScript bytes, over 1.1 MB.

## HTML document

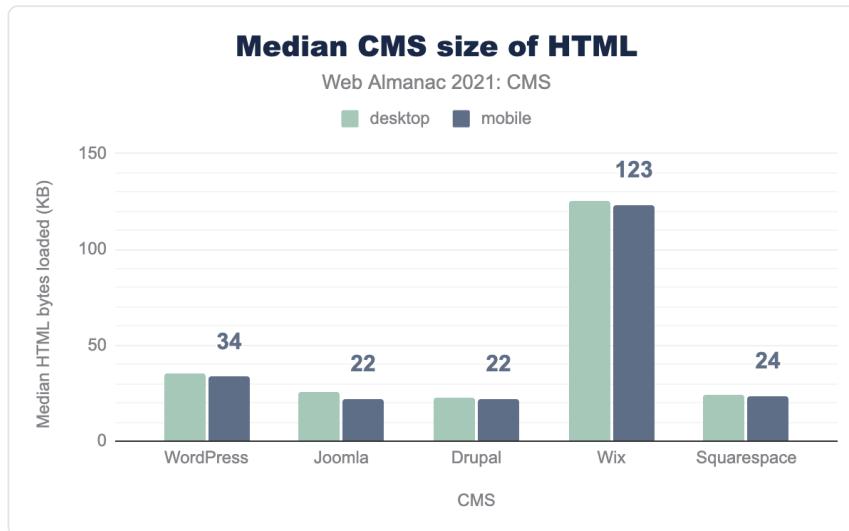


Figure 16.25. Top 5 CMSs median HTML weight.

Examining the HTML document sizes, we can see that most of the top CMSs deliver a median HTML size of ~22 KB–34 KB, except Wix which delivers substantially more HTML of ~123 KB. This can suggest extensive use of inlined resources and shows an area that can be further improved.

## CSS



Figure 16.26. Top 5 CMSs median CSS weight.

Next, we examine the use of explicit CSS resources that are downloaded. Here we can see a different distribution between platforms, strengthening the differences in inlining approaches. Wix delivers the fewest CSS resources, with only ~25 KB sent on mobile views; WordPress delivers the most with ~115 KB.

## Fonts

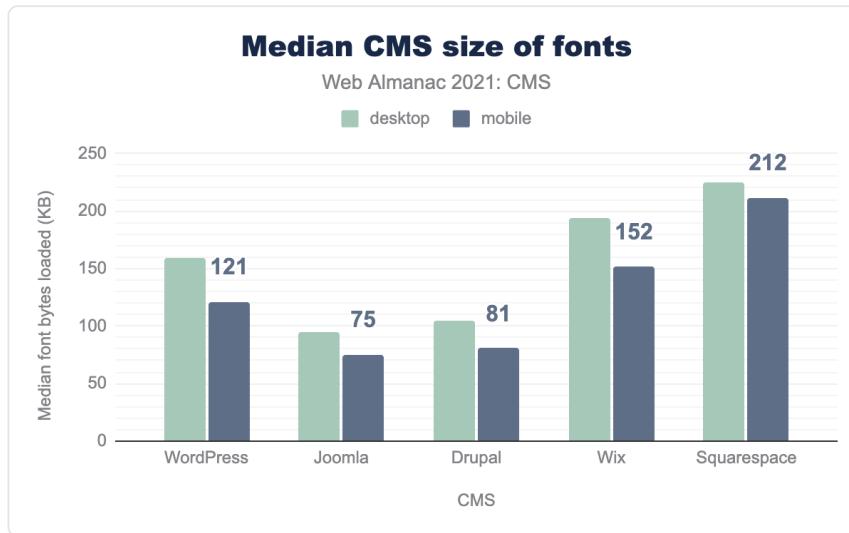


Figure 16.27. Top 5 CMSs median fonts weight.

To display text, web developers often choose to use a variety of fonts. Joomla delivers the fewest font bytes, with 75 KB on mobile views, and Squarespace delivers the most with 212 KB.

## WordPress specific

WordPress is the most commonly used CMS today—almost 3 out of 4 sites built with a CMS are using WordPress, thus deserving further discussion.

WordPress is an open-source project, which has been around since 2003. Many sites built on WordPress use various themes and plugins, sometimes through page builders such as Elementor or Divi.

The 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 websites. This flexibility is important when explaining the market share, but also complicates the discussion around WordPress based site performance.

Contributors from the WordPress community recently acknowledged the current state of performance, in this proposal<sup>795</sup> to create a performance dedicated core team, which can hopefully improve the current performance of the average WordPress sites.

## Adoption

First, we examined WordPress adoption by geography, across all sites in our dataset.

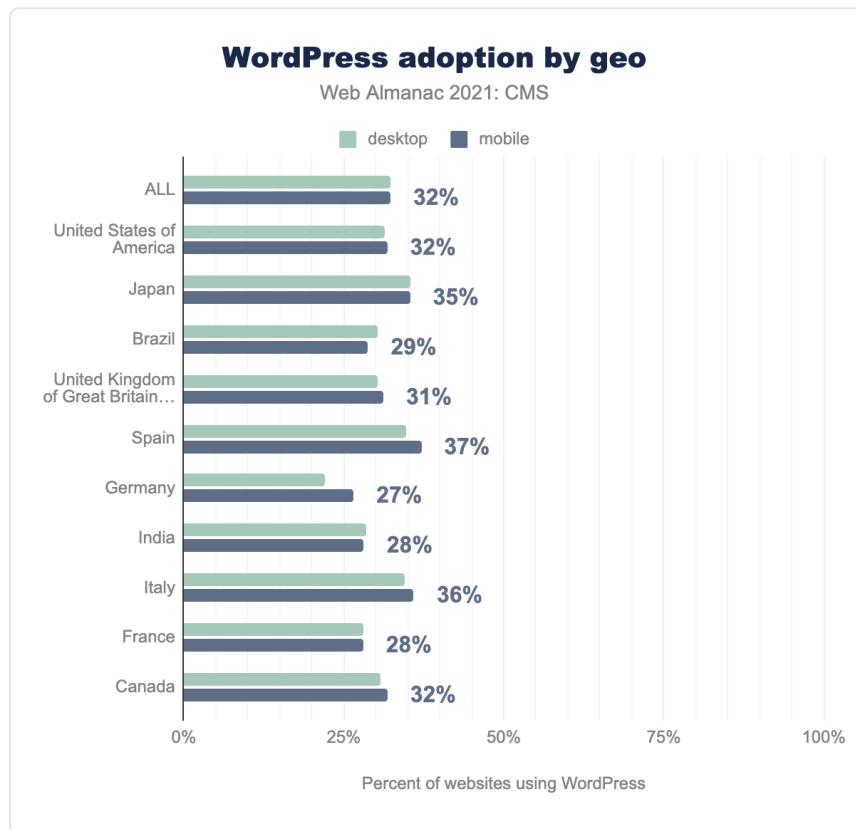


Figure 16.28. WordPress adoption by country.

In the top 10 countries with the most sites in our dataset, WordPress had over 27% adoption. Spain had the highest WordPress adoption among these countries with 37% of mobile pages using WordPress, compared with Germany where only 28% of mobile pages used WordPress.

<sup>795</sup>. <https://make.wordpress.org/core/2021/10/12/proposal-for-a-performance-team/>

## Passing CWVs by geography

Next, let's look at the amount of WordPress origins with passing Core Web Vitals, but this time, breakdown by geography, for mobile devices.

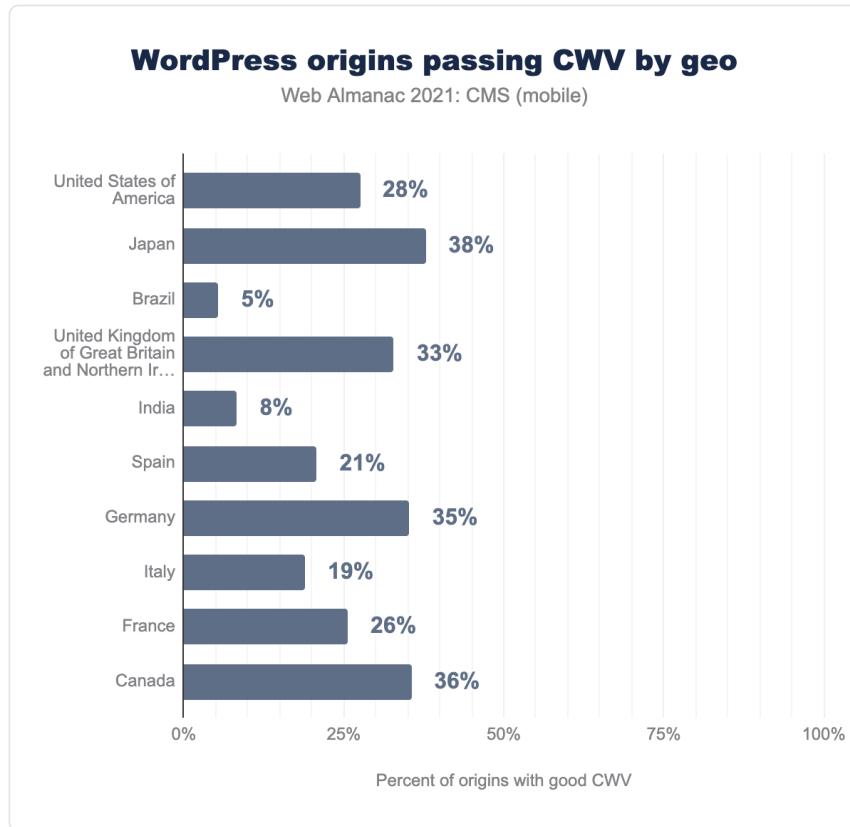


Figure 16.29. WordPress origins passing CWV by geography.

We can see that while WordPress was passing on 19% of the total origins counted across all countries, WordPress sites are passing in a very different percentage in various countries. In Japan, 38% of sites have good CWVs for mobile visitors, but in Brazil, only 5% have good CWVs.

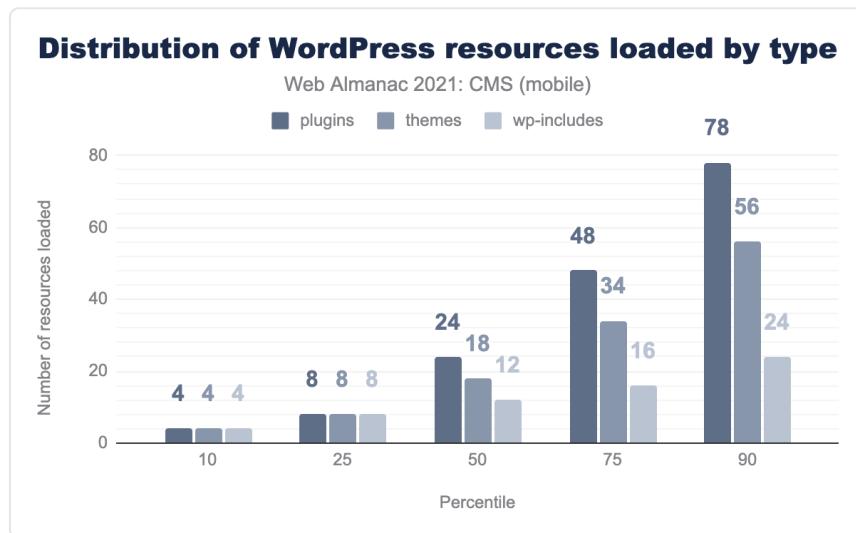
This exposes a very interesting view of Core Web Vitals and hints at a geographical bias when comparing CWV for different platforms. If a CMS only has a presence in certain countries, comparing the aggregate percentage isn't a fair comparison.

WordPress, with a very large adoption around the world, including countries with less powerful devices and slower connections, may suffer from this comparison in some cases, but likely has

room to improve in all geographies. On the other hand, CMSs should strive to offer the best experience in the geography they are targeting, which sometimes means making sites fast enough to work well even under stricter conditions.

## Plugins

We explored how WordPress sites use external resources and separated them between resources that are included in plugins, themes, and shipped in WordPress core (wp-includes).



*Figure 16.30. Distribution of WordPress resources loaded by type.*

The median mobile WordPress page loads 24 resources under the `/plugins/` path, 18 resources under the `/themes/` path, and 12 resources under the `/wp-includes/` path. In the 90th percentile, we see a huge amount of resource requests, with 78 plugin resources, 56 themes, and 24 wp-includes!

WordPress's extension ecosystem provides extraordinary flexibility and may be a major contributor to its high adoption rate. On balance it also appears detrimental to performance in many cases, due to the number of plugins available and the many resources they depend on.

## Conclusion

CMS platforms continue to grow and are becoming more ubiquitous year-over-year. They are

essential for easily creating and consuming content on the internet, especially as more people and businesses establish an online presence.

The introduction of Core Web Vitals, along with the advancements in performance data visibility, has generated a focus on web performance across the web, and we hope these insights will help us all get a better understanding of the current state of the web, ultimately making the web a better place.

CMSs are doing great work and have a huge opportunity to further improve user experiences on the web at scale, by striving to enhance their infrastructure, experiment and integrate with new standards as they evolve, and follow best practices.

On the other hand, Core Web Vitals still have some progress and evolving to do.

We mentioned the thoughts towards a better responsiveness metric<sup>796</sup> above. In addition, navigations between pages in a site should be better tracked and take into account the difference between Single-Page Applications (SPAs) and Multi-Page Applications (MPAs)<sup>797</sup> architectures.

Let's continue pushing forward.

---

## Author

---



Alon Kochba

 @alonkochba  alonkochba  alonkochba

Alon Kochba is a software developer at Wix, where he heads the performance efforts. Alon comes from a back-end background, with extensive experience in networking, and enjoys making the web faster at scale.

---

796. <https://web.dev/responsiveness/>

797. <https://web.dev/vitals-spa-faq>

# Part III Chapter 17

# Ecommerce



Written by Tom Robertshaw

Reviewed by Rockey Nebhwani, Alan Kent, Manuel Garcia, and Fili Wiese

Analyzed by Rajiv Ramnath

Edited by Shaina Hantsis

## Introduction

In this chapter, we review the state of ecommerce on the web. An ecommerce website is an “online store” that sells physical or digital products. When building your online store, there are several types to choose from:

- **Software-as-a-Service (SaaS)** platforms such as Shopify minimize the technical knowledge required to open and manage an online store. They do this by restricting access to the codebase as well as removing the need to worry about hosting.
- **Platform-as-a-service (PaaS)** platforms such as Adobe Commerce (Magento) provide an optimized technology stack & hosting environment while still providing full codebase access.
- **Self-hosted** platforms such as WooCommerce

- There are also **headless** platforms like CommerceTools that are “API-as-a-service”. They provide the ecommerce backend as a SaaS and the retailer is responsible for building and hosting the frontend experience.

Note that platforms may fall into more than one of these categories. For example, Shopware has SaaS, PaaS, and self-hosted options.

## Platform detection

We used an open-source tool called Wappalyzer<sup>798</sup> to detect technologies used by websites. It can detect content management systems, ecommerce platforms, JavaScript frameworks and libraries, and more.

For this analysis, we considered any of the following to indicate that a website is an ecommerce website:

- Use of a known ecommerce platform (see limitations)
- Use of a technology that implies an online store, e.g., Google Analytics Enhanced Ecommerce<sup>799</sup>

You can learn more about the Methodology.

## Limitations

Our methodology has some limitations which affect its accuracy.

Firstly, there are limitations to our ability to recognize an ecommerce site:

- Wappalyzer must have detected an ecommerce platform.
- The detection of a payment processor such as PayPal was insufficient for a website to be considered to be ecommerce. This is because there are sites that accept online payments which are not online stores, e.g., B2B SaaS.
- If the ecommerce platform is hosted within a sub-directory of the website, it cannot be detected as only home pages are analyzed.
- A headless implementation reduces our ability to detect the platform in use. One of

798. <https://github.com/AliasIO/wappalyzer/>

799. <https://developers.google.com/tag-manager/enhanced-ecommerce>

the primary methods to detect an ecommerce platform is to recognize common HTML or JavaScript components. So, a headless website that does not use the ecommerce platform frontend makes it hard to detect as ecommerce.

Next, the accuracy of metrics or commentary may also be affected by the following limitations:

- Any trends seen may be influenced by changes in detection accuracy and not entirely a reflection of industry trends. For example, an ecommerce platform may appear to become more popular because the detection method has improved.
- All website requests were made from the United States. If a website redirects to a more appropriate website based on geographic location, the final location will be analyzed.
- The sites crawled are from the Chrome UX Report which has a bias towards websites visited by users of the Chrome browser.

## Ecommerce platforms

Our analysis considered mobile and desktop websites. These sites are those that are actively visited by Chrome users, see the Methodology for more information. Most of the websites visited are in both result sets but some are only in one. We will often share statistics for mobile and desktop. When there is little variation, we may choose to only show one. In this case, unless otherwise noted, only the mobile metrics will be shown.

The mobile analysis received responses from 7.5 million sites and found that 1.5 million (19.5%) of them had some form of ecommerce functionality. Similarly, the desktop analysis received responses from 6.3 million sites and found that 1.3 million (20.2%) were ecommerce.

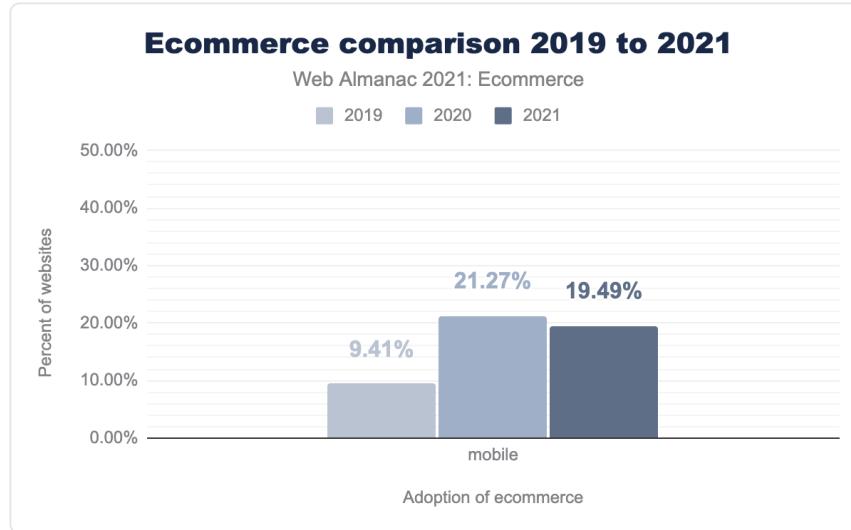


Figure 17.1. Ecommerce comparison 2019 to 2021.

The overall share of ecommerce sites shrunk by 1.8% on mobile (1.6% on desktop) compared to last year's report which found 21.3% of sites were ecommerce (21.7% on desktop). The number of ecommerce sites still increased, with 4.5% more found this year on desktop (8.3% on mobile) compared to last year. However, this growth didn't keep pace with the growth in the overall list of sites visited by Chrome users.

Comparing this with the 2019 results<sup>800</sup> where 9.45% of mobile sites were ecommerce, we can see that while the change in the last year has been insignificant, over the last 2 years the increase is dramatic and sustained.

However, this should not be considered as evidence of ecommerce growth in response to COVID-19. As was reported last year<sup>801</sup>, this increase comes from our improved ability to detect ecommerce platforms: from increased platform coverage, to also using secondary signals such as the presence of Google Analytics Enhanced Ecommerce to indicate that a site is ecommerce.

## Top ecommerce platforms

Our analysis detected 215 ecommerce platforms, a 48% increase in platforms compared to the 145 that were found last year. Despite this, only 10 platforms have greater than 0.1% usage on either desktop or mobile.

<sup>800</sup>. <https://almanac.httparchive.org/en/2019/ecommerce#platform-detection>

<sup>801</sup>. <https://almanac.httparchive.org/en/2020/ecommerce#ecommerce-platforms>



Figure 17.2. Top ecommerce platforms.

WooCommerce<sup>802</sup>, a plugin for WordPress<sup>803</sup>, is the most prevalent ecommerce platform with almost 6% of all websites using it. This represents 30% of the ecommerce market on mobile.

Shopify<sup>804</sup>, a SaaS solution, is the second most popular solution with approximately half as many websites as WooCommerce. It has a 14% share of the ecommerce market on mobile.

PrestaShop<sup>805</sup> is an open-source platform and is the third most used platform at around one-sixth the prevalence of WooCommerce.

4 of the top 10 platforms have open-source and self-hosted editions: WooCommerce, PrestaShop, Magento<sup>806</sup>, and Shopware<sup>807</sup>. We do not detect different versions of platforms, and so cannot distinguish between the open-source and commercial versions of Magento and Shopware.

6 of the 10 platforms are SaaS (or have SaaS versions): Shopify, Wix eCommerce<sup>808</sup>, Squarespace

802. <https://woocommerce.com/>

803. <https://wordpress.org/>

804. <https://shopify.com/>

805. <https://www.prestashop.com/>

806. <https://magento.com/>

807. <https://www.shopware.com/>

808. <https://www.wix.com/ecommerce/website>

Commerce<sup>809</sup>, BigCommerce<sup>810</sup>, Shopware, and Loja Integrada<sup>811</sup>.

Note: There was an issue<sup>812</sup> with the July 2021 HTTP Archive data which resulted in the number of OpenCart<sup>813</sup> sites being under-reported. It is worth acknowledging that in the September results 10,801 OpenCart sites were detected. If a similar number of OpenCart sites were to have been detected in July, it would put it in between BigCommerce and Shopware in terms of popularity.

## Top ecommerce platforms by website popularity

This year, the Chrome User Experience Report<sup>814</sup> provided a popularity rank for each website. This allowed us to break down top ecommerce platforms by their popularity in different segments of the market. “All” refers to all 7.5 million sites that were profiled on mobile and 6.3 million sites for desktop.

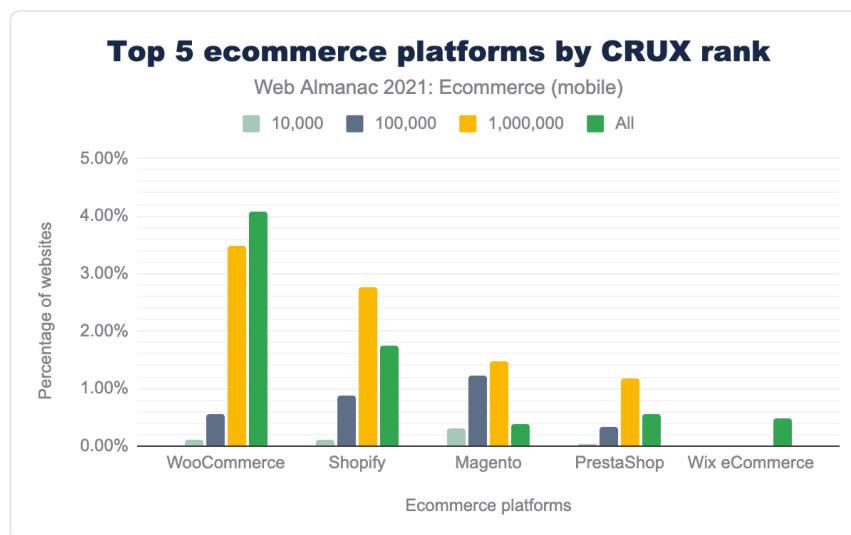


Figure 17.3. Top 5 ecommerce platforms share by CRUX rank

With websites ranked, we can make observations on how platform popularity changes in different segments of the market:

- WooCommerce is the most popular ecommerce platform overall and in the top 1

809. <https://www.squarespace.com/ecommerce-website>

810. <https://www.bigcommerce.com/>

811. <https://lojaintegrada.com.br/>

812. <https://github.com/HTTPArchive/httparchive.org/issues/414>

813. <https://www.opencart.com/>

814. <https://developers.google.com/web/tools/chrome-user-experience-report/>

million.

- Shopify is more popular among websites that are in the top 1 million (as a percentage) compared to all sites analyzed.
- Magento is the most popular of the five shown amongst the top 10,000 sites.
- No Wix eCommerce sites were identified in the top 100,000. Only 164 on mobile were identified in the top 1 million. Almost the entirety of the Wix eCommerce footprint was on sites ranked lower than 1 million.

### Top 1 million sites

Another way to look at the results is to consider the most popular platforms within each tier of rankings. We expected to see different trends among the top tier e.g., top 10,000 sites compared to those within the top 1 million sites.



Figure 17.4. Top ecommerce platforms of 1 million sites

In the top 1 million sites, WooCommerce and Shopify are still the leading platforms with 3.49%

and 2.76% of requests on mobile respectively. However, there's a much smaller gap between them when compared to all sites analyzed. Among all site requests on mobile, WooCommerce was over twice as common as Shopify whereas in the top 1 million it's only 25% more prevalent.

We also see Magento take the third spot over PrestaShop. Wix eCommerce and Squarespace ecommerce are no longer in the top 7 platforms. Instead, we see Shopware, BigCommerce, and Salesforce Commerce<sup>815</sup> ahead of them.

## Top 100,000 sites



Figure 17.5. Top ecommerce platforms of top 100,000 sites

When we consider the top 100,000 sites by CrUX rank the picture changes quite drastically. Magento is now the most popular ecommerce platform vendor with 1.21% of mobile sites. Shopify maintains second place (with 0.88%) while Salesforce Commerce Cloud is third (0.63%). SAP Commerce Cloud<sup>816</sup> rises up the leaderboard to sixth place to show that the enterprise platforms are more competitive in this space.

815. <https://www.salesforce.com/uk/products/commerce-cloud/overview/>

816. <https://www.sap.com/uk/products/commerce-cloud.html>

## Top 10,000 sites



Figure 17.6. Top ecommerce platforms of top 10,000 sites

The share of sites that are powered by an ecommerce platform in the top 10,000 sites is noticeably smaller.

Salesforce Commerce Cloud and SAP Commerce lead and power a similar number of ecommerce sites (0.70 and 0.68% respectively on mobile).

As we continue down the leaderboard, there are few surprises in this space. Quite a way off the top two spots is Magento (an Adobe product) with 0.32% share of the top 10,000 sites. Following that is HCL Commerce<sup>817</sup> (previously known as IBM WebSphere Commerce) and Oracle Commerce<sup>818</sup>. All of these platforms are commonly considered to be well suited to larger enterprises.

817. <https://www.hcltechsw.com/commerce>

818. <https://www.oracle.com/uk/cx/ecommerce/>

## The impact of COVID-19

It is hard to compare the total number of ecommerce sites found across years. As described earlier, this is because the ability to detect whether a site is ecommerce has been improved substantially. In part through the use of secondary signals such as Google Analytics Enhanced Ecommerce integration.

So instead, last year's report focused on a small number of platforms to see how their use had changed. The early signs in the first half of 2020 were that there were measurable and notable increases in Shopify and WooCommerce use. The growth was in the region of 20% between January 2020 and July 2020 while other platforms like Magento did not see the same growth. These platforms are known for their low entry costs and ease of use, while Magento is not.

Fast-forward to 2021, people and businesses around the world have continued to adapt. Ecommerce in the US in 2020 saw revenue growth of 32.4% according to a report<sup>819</sup> by the Commerce Department. In the UK, the Office of National Statistics reported<sup>820</sup> a 46% growth.



Figure 17.7. Ecommerce platform growth Covid-19 impact

We can also look at results on a month-by-month basis between February 2019 and July 2021. However, before conclusions are drawn, it must be noted that sometimes platform detection issues are responsible for changes in market share. One specific issue was the drop in WooCommerce market share between February and June 2021 which was identified as a

819. <https://www.digitalcommerce360.com/article/coronavirus-impact-online-retail/>

820. <https://internetretailing.net/industry/industry/ecommerce-grew-by-46-in-2020--its-strongest-growth-for-more-than-a-decade--but-overall-retail-sales-fell-by-a-record-19-ons-22603>

bug<sup>821</sup>).

With that in consideration, we may still note that on mobile:

- WooCommerce has grown from 3.48% to 5.93%. The majority of this growth occurred immediately following the COVID-19 restrictions that Western countries put in place.
- The rate of growth for Shopify increased significantly during 2020, growing from 1.61% to 2.50% during that year. However, this growth rate has not been sustained.
- Also, during this time, we see Magento, who previously was competing with Shopify, drop below PrestaShop. Moving from 1.25% share of all sites to 0.72%.

In the author's point of view, there was a rapid initial response by small businesses to add an ecommerce channel to their business. This was achieved mostly in the first half of 2020 through the use of cost-effective and easy-to-use platforms such WooCommerce and Shopify.

However, the vast majority of the increased online revenues reported is expected to have benefited those businesses that were already ecommerce-enabled.

## Ecommerce user experience

The objective of an ecommerce site is to generate revenue. A company will adopt multiple strategies to fulfill this objective. At a high level, this might be to offer a feature-rich experience that considers a breadth of buying journeys. They will also want the website to be as fast as possible. It's clear how both of these strategies work towards the objective but they can also work against each other at the same time.

Later, we will look at some of the tools & tactics that are used for creating a feature-rich experience.

First, we will evaluate site technical quality and performance. There is no single metric or tool that can be used to definitively gauge either one, so we drew on multiple:

- Google Lighthouse
- Core Web Vitals from Chrome UX Report
- WebPageTest

---

<sup>821</sup>. <https://github.com/HTTPArchive/almanac.httparchive.org/issues/1843>

## Lighthouse

One way of measuring the technical quality of a web page is with Google Lighthouse<sup>822</sup>. A lighthouse test provides a score out of 100 for each of five categories. The figure below shows the median score for each category across all ecommerce websites requested.

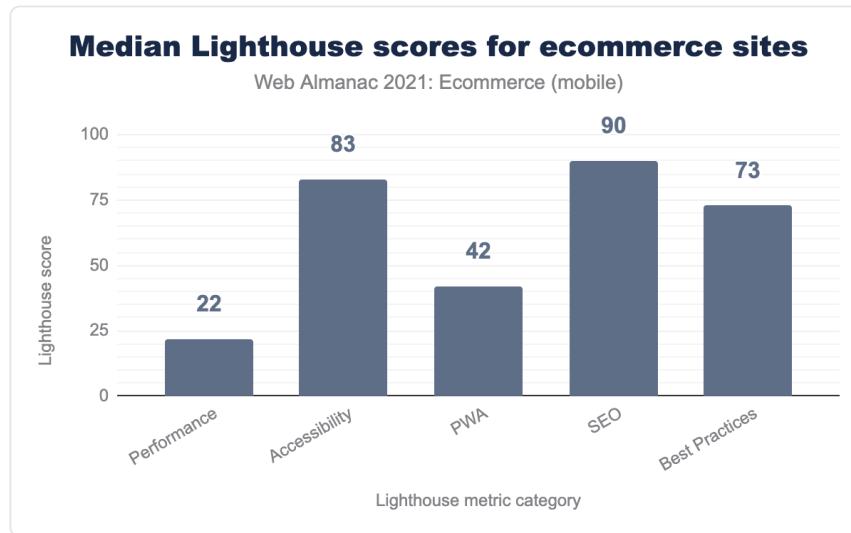


Figure 17.8. Median Lighthouse scores for ecommerce websites

The most important point to note here is that ecommerce sites are struggling to achieve a good lighthouse score for performance. This may be because it takes a greater level of effort to achieve a good score in this category.

## Lighthouse scores by platform

When we broke the Lighthouse scores down by ecommerce platform vendors, there was relatively little variation. This suggests that each ecommerce platform provides similar out-of-the-box capabilities in each of these areas.

### Performance

Performance is an emergent system property; it is not something that you can implement as you would a new feature. It is something that has to be factored into everything you do. One

<sup>822</sup> <https://developers.google.com/web/tools/lighthouse/>

simpistic view is that the more features that you add to your site, the slower it will be.

At the same time, it is now common knowledge that a faster site leads to a higher conversion rate. So why do we see such poor performance scores for ecommerce sites? One reason for this may be that the site speed and conversation rate statistics are always offered without any consideration for the decisions that ecommerce businesses face. When revenue growth is required every year, even the law of diminishing returns says that conversion rate improvements cannot only be met through speed gains. This, together with the high consumer demands on the ecommerce experience leads to a situation where more features become the priority.

What's more, there is often more nuance to the decision to include a feature. For example, do the benefits of a live chat widget outweigh the performance impact? Does the answer change depending on the context? Should you wait for a developer to install it to ensure that it's lazy-loaded or just use Google Tag Manager? What's the opportunity cost of not using that development time for something else?

Another way of viewing performance is that it is a shared resource that suffers from the tragedy of the commons paradigm<sup>823</sup>. It's at its highest level at the start of a project and is depleted over time with requests from different stakeholders that all have a right to consume it.

The best results are likely to be found by those businesses that can find a balance between site speed and user experience. They will minimize the impact of features on the initial page load, while still being able to offer a great user experience.

823. <https://www.investopedia.com/terms/t/tragedy-of-the-commons.asp>



Figure 17.9. Median Lighthouse performance scores for ecommerce websites

The most variation between platforms was found for the performance scores. Shopify and Wix eCommerce were the most performant with a median lighthouse performance score of 27/100 on mobile. The lowest scorers were Loja Integrada with 6/100, Squarespace Commerce with 16/100, and Magento with 18/100. To reiterate, these are all poor scores.

Shopify, to its credit, has recently added a requirement<sup>824</sup> on all new marketplace themes to achieve an average Lighthouse performance score of 60/100. It will be interesting to see how this affects their results in future analyses.

824. <https://shopify.dev/themes/store/requirements>

## Accessibility



Figure 17.10. Median Lighthouse accessibility scores for ecommerce websites

The top 8 platforms score very similarly on the median accessibility metric. We also expect them to improve further as accessibility legislation and awareness increases.

Improvements may come from platforms increasing the accessibility of their standard themes. BigCommerce, for example, has updated the default theme<sup>825</sup> to meet Website Content Accessibility<sup>826</sup> Guidelines (or WCAG) 2.1 Level AA standards.

Platforms can also encourage the wider app and theme communities to provide a high standard of technical quality. Shopify announced<sup>827</sup> a minimum Lighthouse accessibility score requirement for any new marketplace themes.

For more detailed research on accessibility scores across the web, read the Accessibility chapter.

## PWA

It appears that PWA support is not a priority for all ecommerce businesses. We might consider two reasons why this may be the case:

825. [https://support.bigcommerce.com/s/blog-article/aAn4O0000000CdJDSAO/improvements-to-accessibility-coming-in-cornerstone-52?language=en\\_US](https://support.bigcommerce.com/s/blog-article/aAn4O0000000CdJDSAO/improvements-to-accessibility-coming-in-cornerstone-52?language=en_US)

826. <https://www.w3.org/WAI/standards-guidelines/wcag/#intro>

827. <https://www.shopify.com/partners/blog/theme-store-accessibility-requirements>

- There's little research into the consumer adoption of PWA features such as adding to their home screen.
- Safari on iOS does not support the Push Notification API or the ability to add a PWA to the home screen. The significant size of the iOS market share reduces the payoff of investing in PWA.

## Best Practices



Figure 17.11. Median Lighthouse best practices scores for ecommerce websites

Wix Ecommerce achieves the highest median Lighthouse best practice score with 93/100. While it is focused on small businesses and therefore may, on average, provide a simpler user experience it is impressive that it scores so highly.

## Core Web Vitals

In 2020 Google started an initiative under the term Core Web Vitals (CWV) which looked to help website owners and developers focus on three performance metrics that are critical for a good user experience. These metrics are:

### Large Contentful Paint<sup>828</sup> (LCP)

828. <https://web.dev/lcp/>

- Measures *loading* performance. To provide a good user experience, LCP should occur within 2.5 seconds of when the page first starts loading.

### First Input Delay<sup>829</sup> (FID)

- Measures *interactivity*. To provide a good user experience, pages should have an FID of 100 milliseconds or less.

### Cumulative Layout Shift<sup>830</sup> (CLS)

- Measures *visual stability*. To provide a good user experience, pages should maintain a CLS of 0.1. or less.

As Core Web Vitals are now ranking factors in Google's search algorithm<sup>831</sup> they have gained increased attention from ecommerce businesses.

The Chrome User Experience report enables the collection of these metrics from real users. We can therefore consider the results to be more accurate compared to traditional "lab" tests which simulate a page load in a controlled environment.

In this section, we will review sites that have reached a "good" threshold on all three metrics: LCP, FIP, and CLS.

829. <https://web.dev/fid/>  
830. <https://web.dev/cls/>  
831. <https://developers.google.com/search/blog/2020/05/evaluating-page-experience>



Figure 17.12. Real-user Core Web Vitals experiences

Looking at the percentage of sites that have a “good” experience according to CWV by platform, we find that Shopify performs the best with 32.64% on mobile. Whereas only 11.32% of mobile sites on WooCommerce achieve a good experience.

We can compare this to the wider web by looking at the results from the Performance chapter. It found 41% of sites on desktop and 29% of sites on mobile achieved a “good” CWV experience. With this lens, we can say that on average a Shopify store performed better than the average site based on mobile sites, and a WooCommerce site worse. However, it is important to point out that this is correlation rather than causation.

Compared to last year we see an improvement in median CWV scores across all platforms. We find the largest performance improvement was for sites on Shopify. Increasing from 21.24% of sites on mobile having a good CWV experience to 32.64%.

One final point to make is that the percentage of sites achieving a good CWV experience is not correlated with whether a platform is SaaS or self-hosted.

In the next section, we will consider each CWV metric independently to see whether what is

the largest contributor to poor site performance on each platform.

### Largest Contentful Paint (LCP)

Firstly, there is the Largest Contentful Paint<sup>832</sup> which uses the time it takes for the main page content to be loaded as a proxy for how long it takes for the page to be useful.

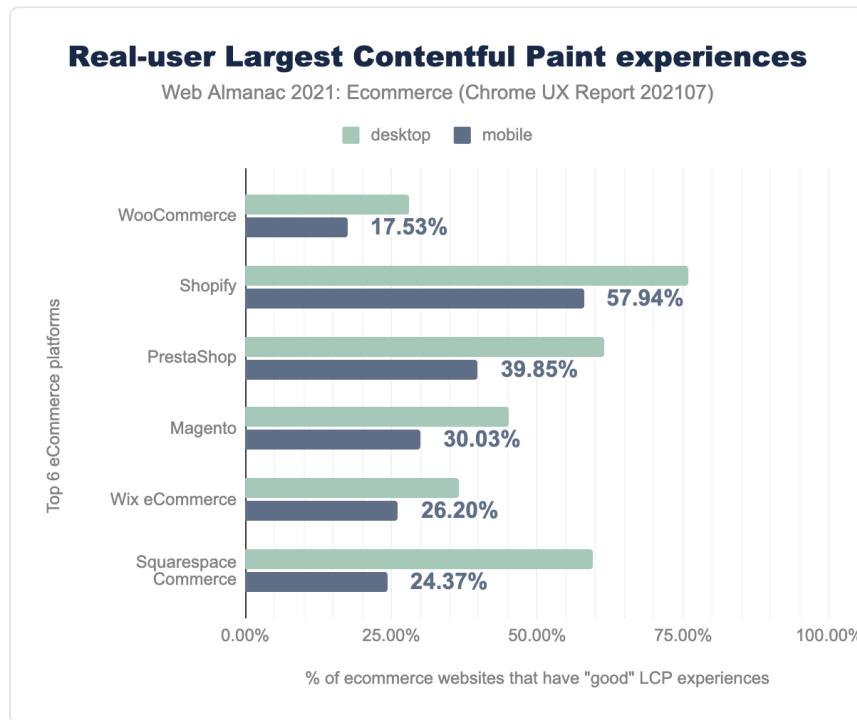


Figure 17.13. Real-user Largest Contentful Paint experiences

Shopify again leads the pack of top ecommerce platforms with 57.94% of Shopify sites on mobile achieving a good LCP experience. Sites that use WooCommerce performed the worst with only 17.53% achieving a good experience. This metric in particular appears to be the largest contributor to WooCommerce poor overall CWV score.

Across the wider web, the Performance chapter found 45% of mobile sites had a good LCP experience. Only Shopify of the top 6 most popular ecommerce platforms achieved better than the average of all sites requested on mobile.

<sup>832</sup>. <https://web.dev/lcp/>

Out of the three CWV metrics, the hosting setup primarily only affects the LCP score. So, at this point, it is worth comparing platforms that are commonly self-hosted against SaaS platforms where infrastructure is managed and optimized by the vendor. We can see that Shopify as a SaaS leads the other platforms. However, the other two SaaS platforms listed, Wix eCommerce and Squarespace Commerce, perform worse on mobile compared to popular self-hosted platforms Magento & PrestaShop.

### First Input Delay (FID)

The second metric, First Input Delay<sup>833</sup>, measures how much work the browser has to do once a website visitor interacts with the site, e.g., clicks on a link or button. It can be seen as a proxy for how responsive the site feels or whether it feels laggy and slow to react to user input.



Figure 17.14. Real-user First Input Delay experiences

Sites on all of the top ecommerce platforms performed well on this metric. On desktop, most of the ecommerce platforms surveyed achieved 100% good FID experience. On mobile, we start

<sup>833</sup> <https://web.dev/fid/>

to see some poor experiences, but the vast majority achieve a good FID experience. Shopify (98.21%) and Squarespace Commerce (98%) perform the best of the top ecommerce platforms with WooCommerce, PrestaShop, and Magento only slightly behind with 98%.

Wix eCommerce is a platform that we've typically seen perform well but FID is one area it falls down on with only 92.05% of its websites having a good FID experience.

That being said, all six perform better than non-ecommerce sites. The Performance chapter found that 90% of all sites on mobile achieved a good First Input Delay experience.

### Cumulative Layout Shift (CLS)

The final of the three CWV metrics is Cumulative Layout Shift<sup>834</sup>. It is a measure of the amount that items on the page “move around”, e.g., a new image appears and pushes the text you were reading or the button you were about to click to a different place.

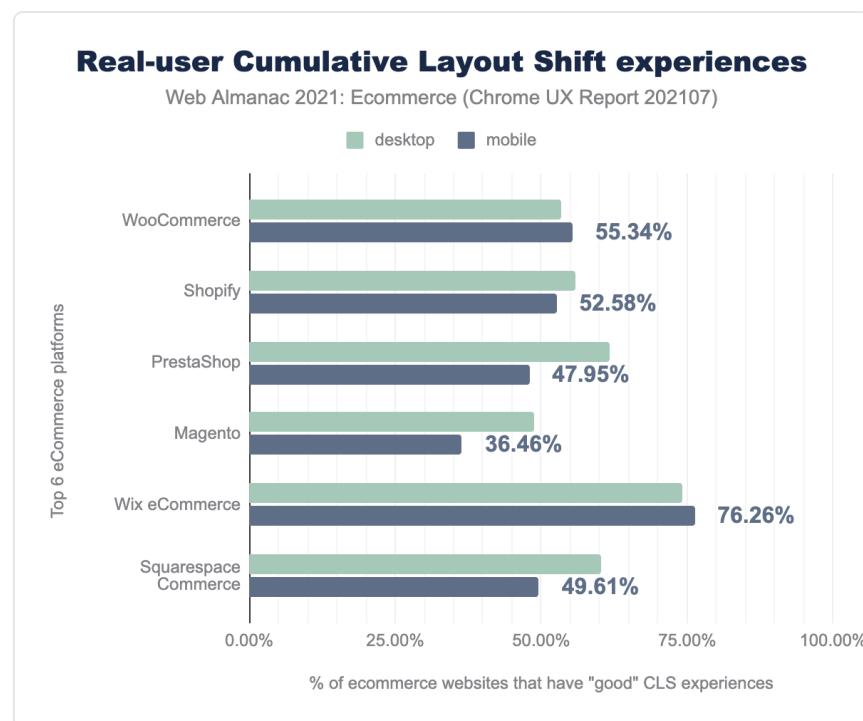


Figure 17.15. Real-user Cumulative Layout Shift experiences

<sup>834</sup> <https://web.dev/cls/>

Of the top platforms, Wix eCommerce outperforms all with 76.26% of mobile sites on the platform achieving a good Cumulative Layout Shift Experience. Whereas less than half as many visitors have a good experience on Magento sites (36.46%).

Comparing these ecommerce sites metrics to the wider web, we see that the top ecommerce platforms perform slightly worse. The Performance chapter found 62% of sites (on mobile and desktop) had a good CLS experience.

## Page anatomy

When it comes to understanding the reasons behind a site's performance, some of the first things that you will look into are the page weight (the number of kilobytes that need to be downloaded), and the number of requests required to load the page.

### Page requests



*Figure 17.16. Page requests distribution.*

The 50th percentile of all ecommerce sites had 101 requests on the homepage on mobile. This is a very similar number to the 98 requests that were found last year. The number of requests per page is very similar across all percentiles when compared to last year.

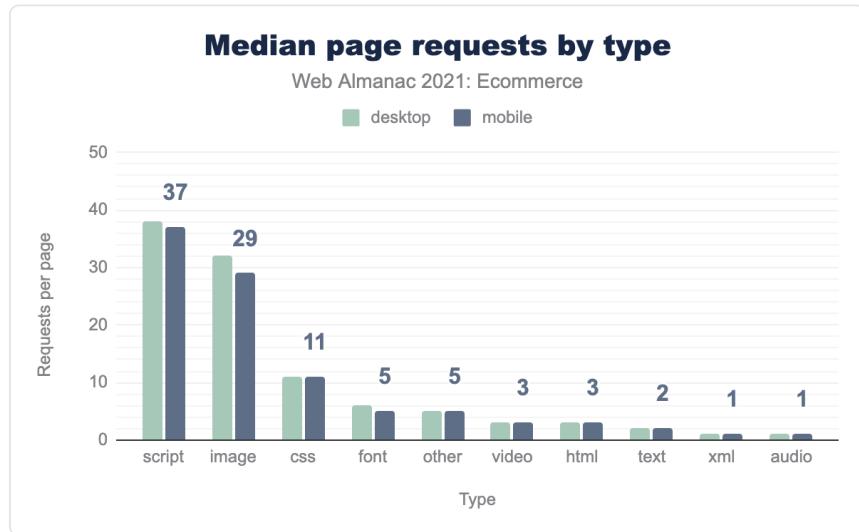


Figure 17.17. Median page requests by type.

Breaking these requests down by type and we can see that JavaScript is the most popular resource to be requested with 37 requests on an average ecommerce mobile homepage. This is a 23% increase from last year where there were 30 JavaScript requests per page. Previously images were the most requested resource with 34 requests per page on mobile, but this is down slightly to 29 requests.

## Page weight

The page weight of a site includes all HTML, CSS, JavaScript, JSON, XML, images, audio, and video.



Figure 17.18. Page weight distribution.

The median page weight of ecommerce homepages was 2.5 MB on mobile. This figure is the same as last year's results, so on average homepages are not getting heavier (or lighter).

The heaviest sites (90th percentile) are 4% heavier than 2020's results so the worst offenders have gotten slightly worse.

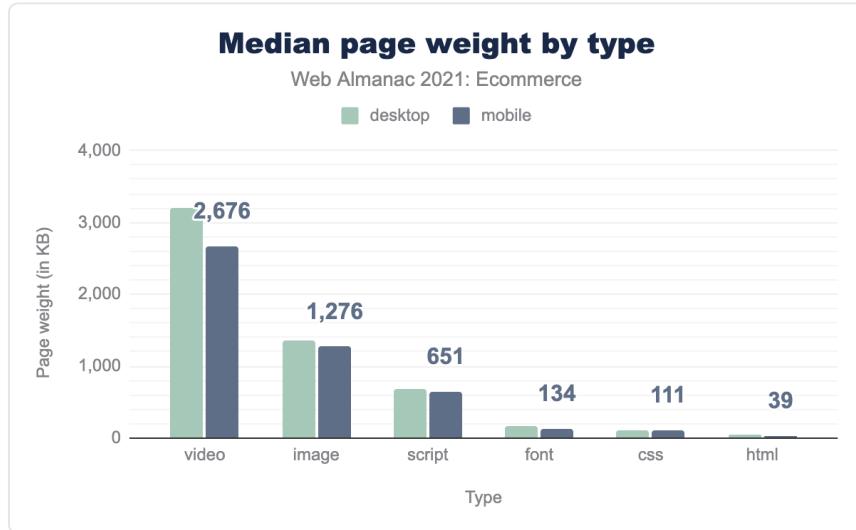


Figure 17.19. Median page kilobytes by type.

To better understand why this might be, we can look at the page weight by resource type. Video is the heaviest resource with 2.6 MB on mobile sites, followed by images (1.2 MB) and JavaScript (0.6 MB). Compared to last year we see a 24% increase in the number of MB of video loaded. Meanwhile, the MBs for all other resource types are steady.

This suggests that the heaviest sites may be those that use video which can quickly increase the overall page weight quite substantially. Given that the median page weight has not changed between 2020 and 2021, this would suggest that the number of sites using video has not changed, but of those that are, they are using it more. An opportunity for further research in this area would be to look at what has caused the video weight increase: are there more videos, are they longer, or higher quality?



Figure 17.20. Page requests by type at 90th percentile.

We saw that the sites with the heaviest pages (17 MB on mobile) were much heavier than the median (4.8 MB). If we look at the page weight by type specifically at the 90th percentile and compare it with the 50th percentile we can see that the weight of all resource types has increased.

The largest contributors to page weight at the 90th percentile continue to be video with 9 MB and images (5.6 MB). It isn't altogether surprising that the heaviest ecommerce homepages are those that use a large amount of video and images. This page is often content-heavy, and these resource types are the most effective way of communicating the brand. While video and images continue to be an important part of the buying experience, in the author's point of view, other page types are unlikely to see these extremes quite as much.

## HTML payload size

The HTML payload is the size of the document response. In addition to HTML, this may include inline JavaScript and CSS.



Figure 17.21. Distribution of HTML bytes per ecommerce page

The median HTML payload was 38 KB on mobile and 39 KB on desktop. While at the 90th percentile, payloads were almost four times larger at 144 KB on mobile and 141 KB on desktop.

Payload size was broadly consistent across both mobile and desktop suggesting that sites are broadly delivering the same HTML to both device types.

## Images

Images are the second most requested resource type as well as the second-largest contributor to page weight.

## Distribution of image requests for ecommerce

Web Almanac 2021: Ecommerce

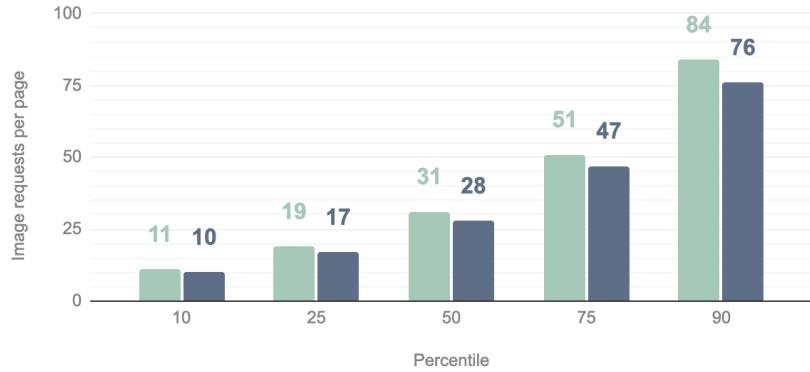
■ desktop    ■ mobile


Figure 17.22. Distribution of image requests for ecommerce

We see the median number of images requested on a mobile homepage is 28, while it is 31 on desktop. 10% of sites load 76 images on mobile, however, this is down from a high of 91 images last year.

Overall, there is a 10-20% reduction in the number of images requested. It is hard to provide a definitive answer, but it may be due to the increased adoption of the lazy loading attribute<sup>835</sup>. As no scrolling or interaction with the site is performed during testing, any assets that are lazy-loaded will not be factored into measurements. Analysis by the JavaScript chapter did find that 17% of sites are using this attribute which gives some weight to this theory.

<sup>835</sup>. <https://web.dev/browser-level-image-lazy-loading/>



Figure 17.23. Distribution of image bytes for ecommerce

If we consider images by weight rather than count, we see a median page weight contribution of 1.2 MB (mobile). At the 90th percentile, this rises to 5.4 MB.

Overall, the weight of images on ecommerce homepages is very similar when compared to 2020's analysis.

Given we have seen that the number of image requests is slightly down, the average weight of each image must have slightly increased.

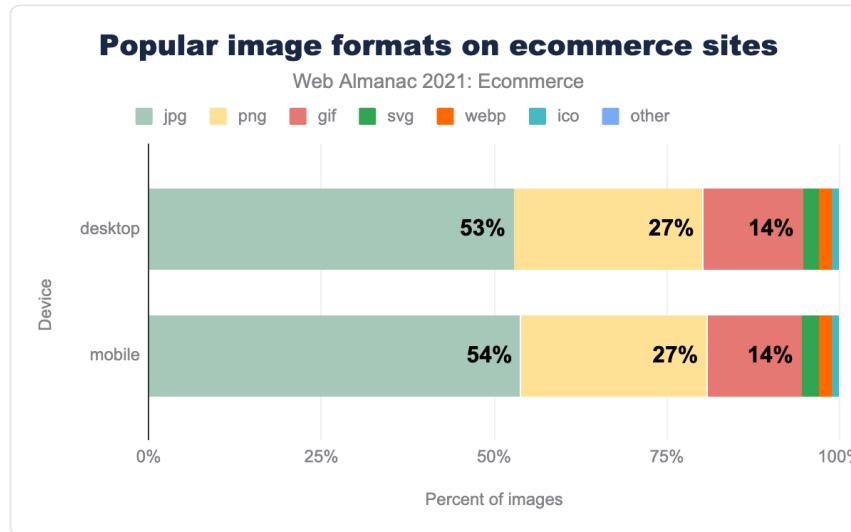


Figure 17.24. Popular images formats on ecommerce websites

Note that some image services or CDNs will automatically deliver WebP (rather than JPEG or PNG) to platforms that support WebP, even for a URL with a `.jpg` or `.png` suffix. For example, `IMG_20190113_113201.jpg` returns a WebP image in Chrome. However, the way HTTP Archive detects image formats is to check for keywords in the MIME type first, then fall back to the file extension. This means that the format for images with URLs such as the above will be given as WebP since WebP is supported by HTTP Archive as a user agent.

The most popular image format was JPG with 54% of images being in this format on mobile. This is an 8% increase on last year when 50% of images were JPGs.

27% of images were PNGs which is a similar proportion to last year. The use of other image types is broadly the same however GIFs have decreased from 17% to 14% on mobile.

Unfortunately, there is still a disappointingly low uptake on WebP support. This is despite it being a more file size efficient format, and is supported in all modern browsers<sup>836</sup>.

## Third-party requests

Ecommerce platforms and sites often make use of third-party content. We use the Third Party Web project to detect third-party usage.

<sup>836</sup>. <https://caniuse.com/webp>

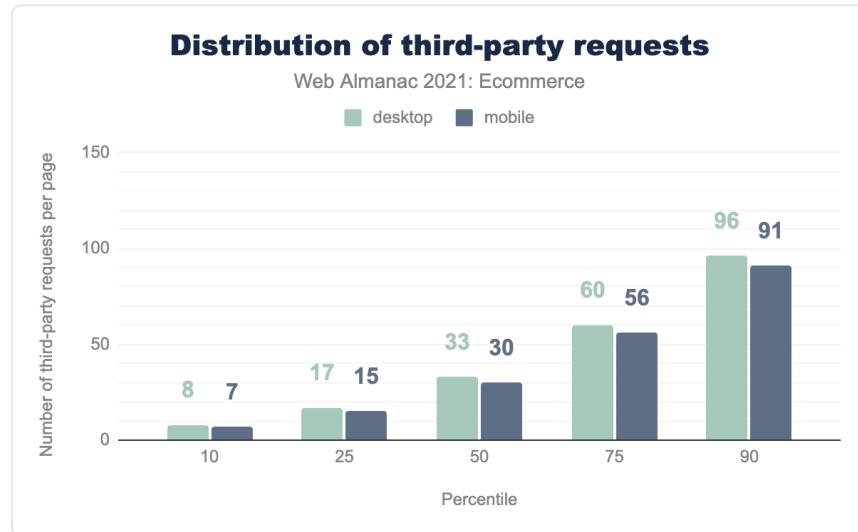


Figure 17.25. Distribution of third-party requests

The median ecommerce site on mobile made 30 requests to third parties. While last year's analysis saw an increase in third-party requests, this year the number is static with little change almost across the board. There is a slight change where the top 10% of pages have reduced the number of third-party requests from 98 to 91 on mobile and 103 to 96 on desktop.

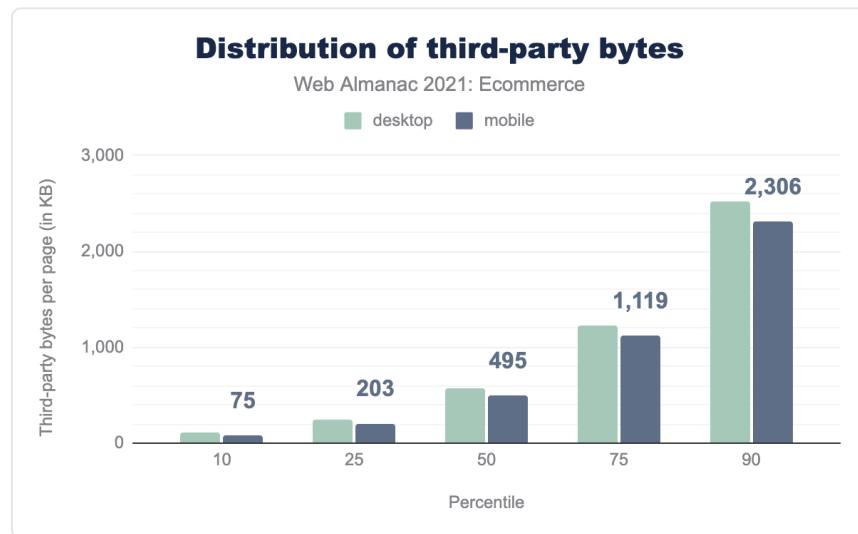


Figure 17.26. Distribution of third-party bytes

The weight of third-party content is also very similar to last year's analysis. With sites in the 50th percentile requesting 495 KB of third-party content. The bottom 10% requested 75 KB while the top 10% requested 2306 KB.

## Tools

In addition to site performance and quality analysis, our Methodology enables us to review other technologies used on ecommerce sites. This provides us with insight into the ecommerce strategies adopted (e.g., internationalization), as well as typical development techniques (e.g., JavaScript libraries used).

### JavaScript frameworks & libraries

Using JavaScript is a popular method of customizing the commerce experience, particularly on SaaS platforms where the core product is a black box.

While we haven't seen a marked increase in the amount of JavaScript used on the ecommerce sites this year, we did want to look into which frameworks and libraries are most commonly used. This may give insight into what JavaScript is being used to achieve.

Unfortunately, we are unable to make statements about the proliferation of headless frontend implementations within ecommerce. One limitation of the methodology is that it is more difficult to detect that a site is ecommerce when it is headless because the typical markers of an ecommerce platform no longer exist. At this point, the analysis falls back on weaker secondary signals.



Figure 17.27. Top JavaScript frameworks on ecommerce sites



Figure 17.28. Top JavaScript libraries on ecommerce sites

We see that jQuery<sup>837</sup> is still the most popular library. Reports of its demise are greatly exaggerated. 93.66% of ecommerce websites profiled were still using it. Many of the popular ecommerce vendors provide jQuery as part of the default frontend. On top of that platforms also live and die by the app and plugin ecosystems where additional functionality can be bought off the shelf. These solutions also regularly use jQuery to provide functionality cost-effectively.

Noticeably GSAP<sup>838</sup> (GreenSock Animation Platform) is included on 15% of ecommerce websites requested on mobile. That's more common than Fancybox<sup>839</sup> (12.48%), a popular Lightbox library, and Slick<sup>840</sup> (9.90%) a library used for creating carousels.

We recognized in the limitation section that the results are going to be skewed because all requests are made to the homepage. This means that the analysis won't find any libraries used for the product detail page media gallery where Slick may have proven even more popular.

837. <https://jquery.com/>

838. <https://greensock.com/gsap/>

839. <https://fancyboxapp.com/docs/ui/fancybox/>

840. <http://kenwheeler.github.io/slick/>

## Analytics

One of the beauties of ecommerce is that you can measure how well you're doing by how many people you convert after they visit the site. In theory, every change you make, every new pricing offer, every new feature can be assessed objectively with analytics.

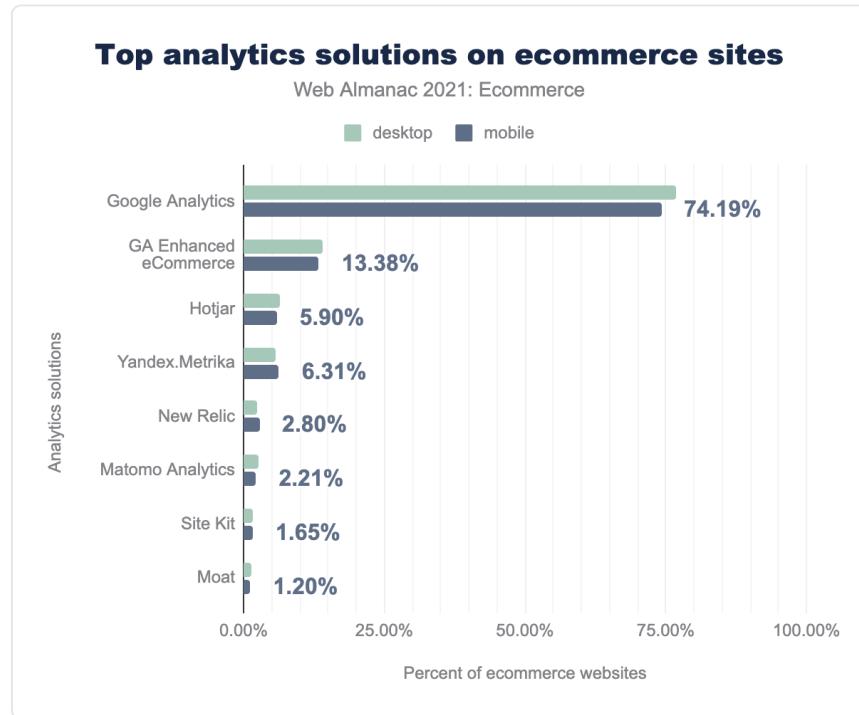


Figure 17.29. Top analytics solutions on ecommerce sites

Google Analytics<sup>841</sup> is the most popular analytics tool, found on 74.19% of websites (mobile). Bemusedly, only 13.38% of mobile requests and 13.99% of desktop requests noted the use of enhanced ecommerce<sup>842</sup>. However, as the main enhanced ecommerce features are for tracking the ecommerce journey through product listing page, product detail page, cart, and checkout, perhaps the reason that we do not see a greater percentage is due to a limitation of the survey being restricted to home pages.

841. <https://marketingplatform.google.com/about/analytics/>

842. <https://support.google.com/analytics/answer/6014872?hl=en#zippy=%2Cin-this-article>

## Tag managers

These tools provide ecommerce and marketing teams with reduced cycle time for launching new features as they allow JavaScript changes to be made to the site without a core website platform deployment (or indeed developer involvement).



Figure 17.30. Top tag managers on ecommerce sites

Google Tag Manager<sup>843</sup> is by far the market leader with 56.39% usage on desktop and 53.95% on mobile. In second and third places were Tealium<sup>844</sup> (0.26% mobile) and Adobe Experience Platform Launch<sup>845</sup> (0.20% mobile).

## A/B Testing

In a similar vein to analytics, implementing an A/B testing solution enables hypotheses to be tested. Providing a feedback mechanism for new features is the only way to understand which strategies are working and which should no longer be invested in.

843. [https://marketingplatform.google.com/intl/en\\_uk/about/tag-manager/](https://marketingplatform.google.com/intl/en_uk/about/tag-manager/)

844. <https://tealium.com/>

845. <https://business.adobe.com/uk/products/experience-platform/launch.html>

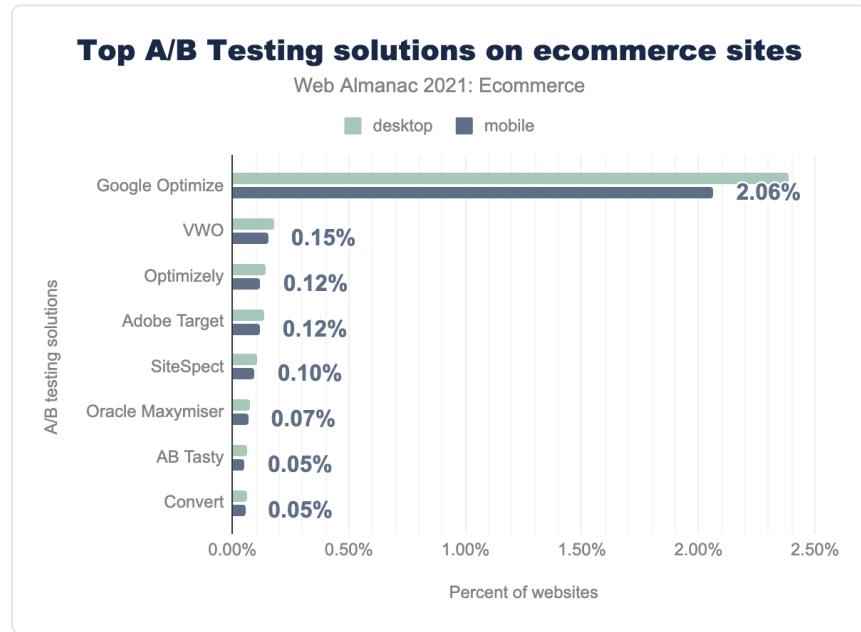


Figure 17.31. Top A/B testing solutions on ecommerce sites

Google Optimize<sup>846</sup> is the most popular A/B testing tool in use on 2.06% of mobile ecommerce sites. VWO<sup>847</sup> was the second most common solution but was found on less than one-tenth the number of sites compared to Google Optimize (0.15% on mobile).

The obvious yet disappointing conclusion is the majority of ecommerce sites were not running A/B tests at the time of the survey.

## Web push notifications

Once a visitor gives their permission, the *Push API* enables ecommerce sites to send push notifications even when the website is not open.

We tried to look at the adoption of web push notifications by ecommerce sites using the Chrome User Experience report. As this is generated from real user data, we can also see the approval rates for push permission requests. Please refer to this Google article<sup>848</sup> for more details on how this data is captured and what metrics are available.

846. <https://marketingplatform.google.com/about/optimize/>

847. <https://vwo.com/>

848. <https://developers.google.com/web/updates/2020/02/notification-permission-data-in-crux>

# 0.43%

*Figure 17.32. Percentage of ecommerce sites using Web Push Notifications (mobile).*

Only 0.43% of home pages on mobile (0.48% on desktop) requested the use of the Web Push API. While, notably, Safari on iOS does not support the Push Notifications API, there is still wide adoption in other browsers. Suggesting there is still a good opportunity to progressively enhance experiences with push notifications at appropriate points in the ecommerce journey, e.g., order updates.

What's more, usage has measurably decreased since last year when 0.69% of mobile sites requested permission to send Push notifications (0.68% on desktop).

We may explain away the low usage statistics by saying that it is from a lack of awareness. However, the reduction in usage suggests a different trend; over a third of sites no longer use push notifications. This may be due to their poor push notification acceptance rates.



*Figure 17.33. Web Push Notification acceptance rates*

The Push notification acceptance rates are very similar to last year's results. The median acceptance rate of push notification requests was 14.23% on mobile. Unfortunately, if there is any trend across year's, it's downwards. At the 90th percentile last year 36.9% of push requests were accepted compared to 29.80% this year on mobile.

The author can offer multiple suggestions as to why the uptake is so low:

- The request is being made at the wrong time, e.g., initial page load, or
- It is made before sufficient motivation has been offered, e.g., without any prompt as to the benefits of accepting notifications, or
- Perhaps more simply that visitors are simply still unaccustomed to web-based push notifications.

## Accessibility overlays

Making your website accessible should not be an afterthought. However, there is an increasing number of technologies that claim to make your website more accessible. An accessibility overlay is JavaScript that tries to apply automated accessibility fixes to the site. They are typically not<sup>849</sup> recommended<sup>850</sup> by accessibility experts.

**0.77%**

Figure 17.34. Percentage of ecommerce sites with accessibility overlays (mobile).

In our research, we found that less than 1% of websites had third-party accessibility tools on their homepage.

Further information on such tools can be found in the Accessibility chapter.

## AMP

**0.61%**

Figure 17.35. AMP usage on ecommerce sites (mobile).

AMP from Google is commonly used within the media industry for providing the latest information fast, but it has struggled to take off in ecommerce. This year we reported less than 0.7% of websites declared AMP compatibility or linked to AMP resources.

849. <https://www.a11yproject.com/posts/2021-03-08-should-i-use-an-accessibility-overlay/>

850. <https://overlayfactsheet.com/>

## Consent management

# 6.85%

Figure 17.36. Third-party consent management solution usage on ecommerce sites (mobile).

The EU Cookie policies and GDPR have increased the complexity of requested marketing permission. This year, we saw 6.85% of ecommerce websites on mobile deploying a third-party consent management app to facilitate collecting consent according to legislation (6.52% on desktop).

## Content Security Policies

On a site where a customer is expected to share sensitive information, it is even more important to have confidence that there is no nefarious code that has made its way into the system. Content Security Policies (CSPs) are a technique to monitor or block requests to third party websites that aren't on a whitelist.

As with many security policies, this form of control can be seen as the antagonist of ecommerce businesses that wish to move quickly with tools such as tag managers whose primary purpose is to add third-party code to sites quickly. In the author's experience, the overhead in managing CSPs has resulted in little usage.

# 23.28%

Figure 17.37. Percent of mobile ecommerce pages that use a Content Security Policy.

On initial reading, we were surprised to find that 25.02% of requests on desktop and 23.28% of mobile pages made use of a Content Security Policy. However, some ecommerce platform vendors provide a lax content security policy out of the box. For example, Shopify sites have a policy that blocks a site from being loaded within an iframe, as well as ensuring all requests are over HTTPS. Without further research, we have not been able to identify how many ecommerce sites are using CSPs as a form of control of third-party assets. Given that only 0.70% of sites are using the "Report Only" mode of CSP which is aimed at testing policy changes before they are enforced, it is likely that very few are.

## Internationalization

A key growth strategy of successful ecommerce businesses is moving into new countries. To do this well, you would want to provide localized language versions of your site.

In this year's analysis, we looked for `hreflang` headers and link tags to see how many sites were using them. These tags are not available out of the box on the most popular platforms (e.g., WooCommerce, Shopify, Magento), the existence of any suggests there would be more than one.

A `hreflang` attribute is used to communicate the language that the page is targeting. Optionally it can also narrow this recommendation to a particular country, e.g., `en-gb` for English targeting Great Britain, as opposed to `en-us` for English targeting the United States.

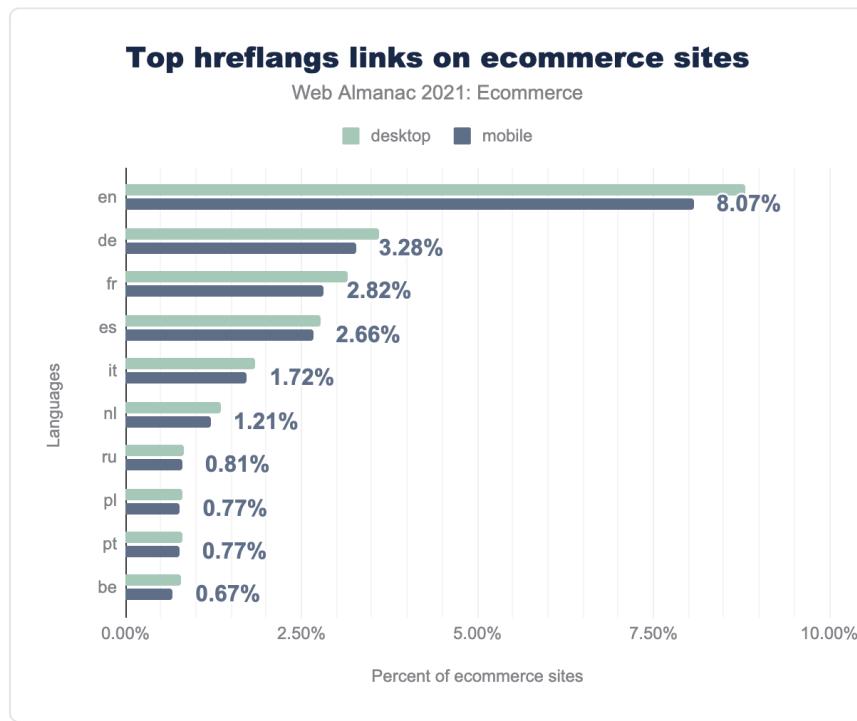


Figure 17.38. Top `hreflang` links used on ecommerce sites

The results identified 8.81% of requests on desktop to specify an English hreflang and 8.07% on mobile ecommerce sites. The next most popular languages were German (3.28% on mobile), French (2.82%), and Spanish (2.66%).

It is hard to draw too many conclusions from this data without further research. However, we can say that it is still uncommon for ecommerce businesses to provide language-specific site variations. Of those that do, they are most likely to declare support for one or more languages used by Western European countries. In the author's experience, the geographic proximity of each of the UK, France, Germany, Spain, and Italy makes internationalization an attractive growth strategy.

Further research could be performed here to better understand the internationalization capabilities of ecommerce websites. For example, looking into the average number of `hreflang` attributes declared may help determine the breadth of multi-region support.

Cross-referencing `hreflang` use with ranking data available from the CRUX metrics could uncover trends of when businesses invest in multi-region support.

## Conclusion

There was a measurable increase in the proportion of sites with ecommerce functionality during Q2 and Q3 of 2020. This growth rate has not been maintained through to 2021. In fact, the percentage of ecommerce sites decreased from 21.27% to 19.49% on mobile suggesting that ecommerce has not grown at the same pace as the wider web.

WooCommerce and Shopify are the most popular ecommerce platforms. They also saw the largest proportion of the growth in response to the pandemic.

For the first time, our analysis benefited from website popularity ranking data. This enabled the review ecommerce platform popularity at different business sizes. In particular, within the 100,000 sites Magento is the most popular platform. It is followed by Shopify and Salesforce Commerce Cloud.

Finally, in terms of site performance, Core Web Vitals has been a prominent industry discussion over the last year because it is now a Google search engine ranking factor. We have seen 10-20% more sites achieve a good CWV on mobile across most of the top 5 platforms. Shopify sites had the highest percentage of good CWV experiences at 33% on average. Despite this improvement since last year, ecommerce sites still perform very poorly across all platforms for Core Web Vitals.

## Future analysis opportunities

One of the methodology limitations is that only the homepage is tested. On an ecommerce site, there will likely be some technologies that are not detectable site-wide, e.g., payments and shipping providers will likely only be visible during the checkout process. This is likely to be

impractical to achieve given the necessary steps to get to this stage of the checkout process.

Evaluating only the homepage also affects our ability to analyze site performance. Arguably the product listing and product detail pages are more important to optimize for speed. Fetching more than one page per site is being investigated<sup>851</sup> and may be available for future editions of the Web Almanac.

Wappalyzer tracks over 2,700 popular web technologies which already provides us with incredible analysis opportunities. However, there is a very long tail of technologies, particularly in ecommerce. At the current time, it's not practical to review categories of technologies within ecommerce, e.g., top personalization tools, top review apps, or top abandoned cart as there isn't enough coverage. This is partly due to the number of technologies that can be detected and partly due to only requesting a single page per site.

As further technologies get supported by Wappalyzer, we may reach a point where further analysis can be done that looks to see if there's any correlation between technology usage, performance, and the CrUX rank of a website.

## Author



**Tom Robertshaw**

🐦 @bobbyshaw    💬 bobbyshaw    💬 tomrobertshaw    🌐 <https://www.space48.com>

Tom is Innovation Director at Space 48<sup>852</sup>, an ecommerce agency for ambitious retailers. He has over a decade of experience in ecommerce working with brands such as Ordnance Survey, Betty's & Taylors of Harrogate and Smythson. He is now leading an initiative to launch a suite of apps for merchants on BigCommerce.

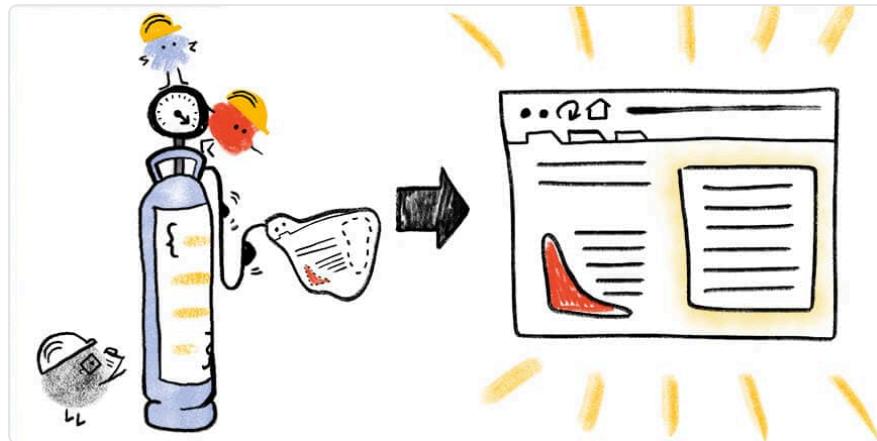
<sup>851</sup> <https://github.com/HTTPArchive/httparchive.org/issues/400>

<sup>852</sup> <https://www.space48.com>



# Part III Chapter 18

# Jamstack



Written by Artem Denysov

Reviewed by Alba Silvente Fuentes, Thom Krupa, and Barry Pollard

Analyzed by Artem Denysov, Barry Pollard, and Rick Viscomi

Edited by Barry Pollard and Shaina Hantsis

## Introduction

*Jamstack has revolutionized the way we think about building for the web by providing a simpler developer experience, better performance, lower cost and greater scalability.*

— Jamstack.wtf<sup>853</sup>

Jamstack stands on JavaScript, API, and Markup architecture. These 3 foundations are decoupled, and the Jamstack site can be built purely using markup. Using pure HTML is “kinda” Jamstack, but it’s really hard to scale. Lucky for us, there’s a huge ecosystem of Static Site Generators (SSGs).

<sup>853.</sup> <https://jamstack.wtf/>

### JavaScript based SSGs:

- Next.js
- Gatsby
- Nuxt.js
- etc

### Traditional:

- Eleventy
- Hugo
- Jekyll
- Hexo
- etc

And there are many more SSGs beyond these<sup>854</sup>. They allow building sites converted to “pure” HTML and JavaScript goodness if needed.

For more complex sites, data has to be structured. There are several ways to store and manage data using headless CMSs<sup>855</sup> via APIs.

Moreover, Jamstack sites need support for server interactions such as form submissions or user input processing. Services like Netlify provide serverless functions<sup>856</sup> support to address this need.

The goal of this chapter is to identify what are the main SSGs used on Jamstack and look at the adoption of Jamstack technology year over year. We looked at how they are distributed around the world, the level of performance of Jamstack sites, and how it is growing. We also explored data of different CDN providers for Jamstack sites. Additionally we dived into results of resources used for Jamstack sites and their impact on user experience.

It's worth mentioning some data disclaimers to consider when reading this chapter:

1. HTTP Archive data of detected SSGs is based on Wappalyzer technology, which has some limitations. It can't detect whether the site was built with certain SSGs such as Eleventy. Also, it can't detect if the site was generated by Next.js Static Rendering<sup>857</sup>

854. <https://jamstack.org/generators/>

855. <https://jamstack.org/headless-cms/>

856. <https://www.netlify.com/products/functions/>

or Server Side Rendering<sup>858</sup>.

2. In our analysis, we can't get any info related to headless CMSs, hence we will not cover this either.
3. We visualize SSG data using top 5 used SSGs based on number of sites built with these SSGs.

More information can be found in the methodology selection.

## Adoption of SSGs

SSG adoption is growing in general by 2x in year over year. In 2019 it was just 0.4% mobile and 0.3% desktop sites. In 2020 the number almost doubled, to 0.6% on mobile and 0.7% on desktop sites. In 2021 they have grown again: 1.1% of mobile and 0.9% of desktop sites. That underlines the trend of that technology. For example, this year Vercel raised a \$102M in series C round<sup>859</sup> and a further \$150M in round D<sup>860</sup> of investment to build a better web with modern technologies like Next.js. Jamstack oriented CDN provider Netlify raised \$105M in their series D<sup>861</sup> of investment. Hence, it's expected that numbers of Jamstack adoption will grow even higher next year.



Figure 18.1. SSG adoption year over year.

857. <https://nextjs.org/docs/basic-features/pages#static-generation-recommended>

858. <https://nextjs.org/docs/basic-features/pages/server-side-rendering>

859. <https://vercel.com/blog/series-c-102m-continue-building-the-next-web>

860. <https://vercel.com/blog/vercel-funding-series-d-and-valuation>

861. <https://www.netlify.com/press/netlify-raises-usd105-million-to-transform-development-for-the-modern-web>

In 2020 the amount of desktop websites increased 2.76 times, while mobile just 1.5 times. In 2021 mobile availability for SSGs built sites became way better than in 2020, and this year there are ~1.9 times more sites than 2020.

## Which SSGs are the most popular

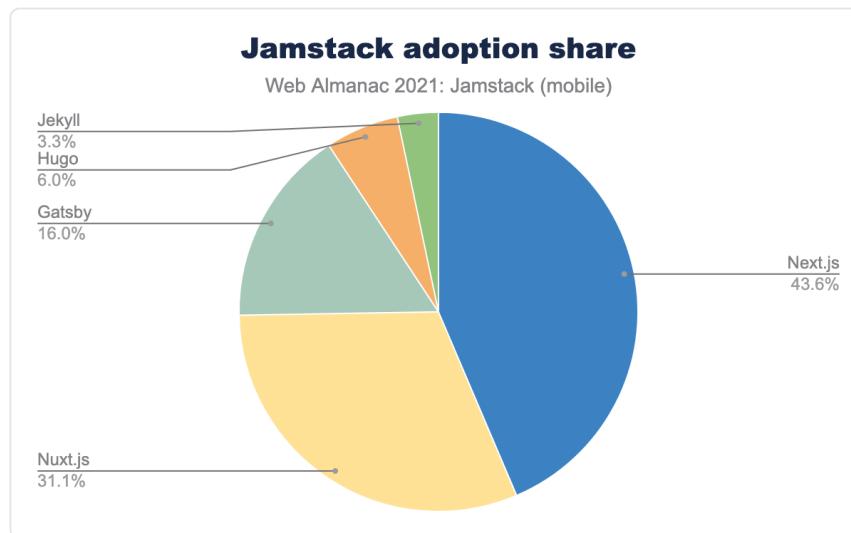


Figure 18.2. SSG adoption share.

Let's begin with understanding which SSG is most popular. Next.js covers 43.6% of Jamstack sites. Nuxt.js is in second place with 31.1%, third is Gatsby with 16.0%, followed by Hugo at 6.0%.

*Please note the original publication of this chapter had different figures due to incorrect over-counting of Nuxt and Next sites. This has been corrected in above figures and, to a lesser degree, in other figures in this chapter.*

All top 3 SSGs are JavaScript based: Next.js and Gatsby use React.js<sup>862</sup> at its core and supplements this by adding their own functionality on top of it. Nuxt.js is based on Vue.js<sup>863</sup>. Having these popular front-end frameworks with huge ecosystems out of the box makes development way easier. Node.js<sup>864</sup> allows JavaScript to run on the server as well as the browser where it has traditionally been used, enabling developers stick to one language. That makes

862. <https://reactjs.org/>

863. <https://vuejs.org>

864. <https://nodejs.org/en/>

adopting these SSGs easier from a server perspective, comparing to Hugo which is based on the Go programming language<sup>865</sup>, and Jekyll based on Ruby<sup>866</sup>.

We will take a look what's the adoption rate of SSGs among web sites.

## Adoption by rank

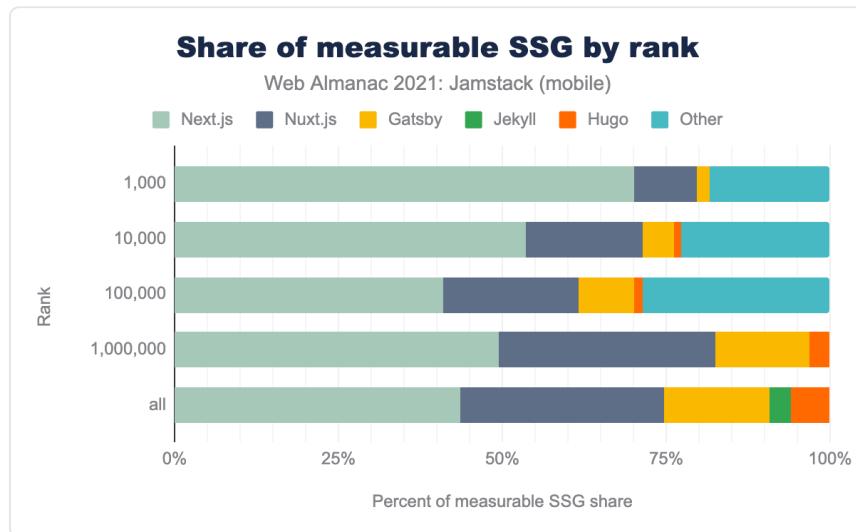


Figure 18.3. SSG adoption share by rank.

Next.js remains a popular SSG across all ranks, but for the top 10k in particular.

## Geographic adoption

In this section we will cover geographic adoption for Jamstack and explore distribution over countries and regions.

## Adoption by country

SSGs are heavily used around the world. The figure below shows the top 10 countries with the highest number of sites.

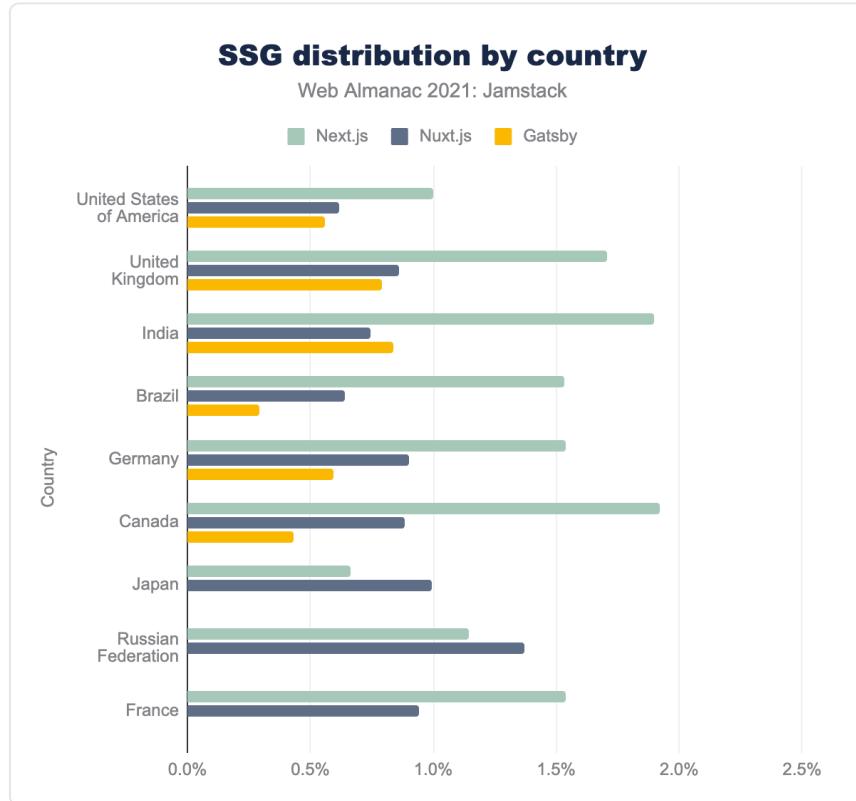
865. <https://go.dev/>

866. <https://go.dev/>



*Figure 18.4. SSG adoption by country.*

In the USA, between 1.2 and 1.4% of all sites pages (which is about 22k pages for desktop and 16k for mobile), are created with SSG. India has a lower number of pages, with just 6k for desktop and 7k for mobile, but 1.7% of all pages is covered by Jamstack technologies. In third place is the United Kingdom, which also has 1.7% of pages.



*Figure 18.5. SSG distribution by country.*

USA has a larger Next.js adoption compared to Nuxt.js and Gatsby. It trends similarly in almost all countries. In most countries, Next.js is a preferable choice. Interestingly Gatsby has no data for 3 of the top 10 countries using Jamstack technologies, but in 2 of them Japan and Russian Federation Nuxt.js is more preferable.

## Adoption by region

We also looked at the adoption levels by regions.



Figure 18.6. SSG distribution by region.

The number of sites in Europe for desktop is 23k versus mobile 26k, which is 1.1% of all web sites in that region. In the Americas, there are 26k desktop sites and 24k mobile sites (1.2% of sites). Asia has almost the same numbers with 21k desktop and 22k mobile as the leading region with greater Jamstack adoption at 1.45%. Oceania and Africa have way lower overall numbers, but they have way greater Jamstack adoption. Oceania 2.19% and Africa 2%. Overall site adoption is at 1.1%.

## Adoption by subregion

We can further break down by subregions to observe additional trends.



Figure 18.7. SSG distribution by sub region.

The list is ordered by the total number of SSG sites, but shows those as a percentage of all sites in that region. It's no surprise that top of the list is Northern America as most companies who invented SSGs are in the USA. However, as a percentage of all sites they are in lower regions with only 1.1% of sites having adopted Jamstack. But surprisingly, Western Europe is in second place and has a similar low percentage adoption compared to some of the sub regions further

down the list.

The tail also shows great results. Subregions with lower number of sites in general adopt technology at a broader based, for example, 4.8% of Micronesia sites.

## SSGs distribution among CDN providers

We described how SSGs are adopted in different countries, so let's analyze which SSG is most popular among different CDN providers.

The 7 most popular CDN providers for SSGs are:

- Netlify
- Vercel
- Cloudflare
- AWS
- Azure
- Akamai
- GitHub

Jamstack CDN services are not just for network delivery. They provide a lot of functionality to allow developers to easily deploy and manage Jamstack sites. For example, Netlify provides easy-to-use functionality to deploy sites in the scope of their service so developers can just update the code and the continuous deployment process is managed for them. Jamstack CDNs provide many other features<sup>867</sup> such as *serverless functions, A/B testing* etc.

On the other hand, Cloudflare, Akamai, AWS are not only used purely for content delivery either, but can also provide protection services, DNS balancing, and more. However, since we can't detect how exactly Cloudflare, Akamai, and AWS are used, results could be false positives if we look at them as Jamstack enablers. The "Jamstack" part could be handled on origin servers and so not actually on these services.

867. <https://bejamas.io/compare/netlify-vs-vercel/>



Figure 18.8. SSG distribution over CDN.

Next.js, is the most popular, mostly served by Cloudflare, Vercel, and AWS. Most of Gatsby's sites use Netlify, AWS, and Cloudflare. Nuxt.js sites preferred to be served by Cloudflare, AWS, and Netlify. Hugo mostly uses Netlify, and it's no surprised that Jekyll is used mostly on GitHub.

On the following graph we show the relative split of CDNs used for popular CDNs:

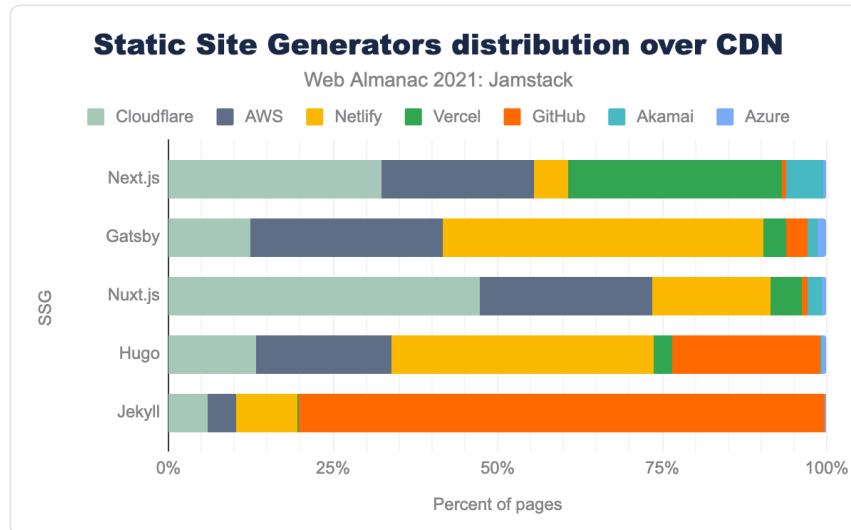


Figure 18.9. SSG distribution over CDN.

Next.js is mostly served by Vercel (the company that invented Next.js). We can see that more generalized CDNs like AWS are not serving significant percentages of Jamstack sites, as opposed to more Jamstack-focussed services like Netlify and Vercel.

GitHub as CDN provider might seem unusual, but *GitHub Pages* allow users to deploy sites on [github.io](https://github.io) subdomains built in Jekyll SSG.

## User experience and performance

In our analysis we wanted to explore what the user experience for the 1.1% of sites that have adopted Jamstack technology. We looked at Lighthouse and Core Web Vitals results.

### Lighthouse

All Lighthouse scores are simulated testing data from our crawl. Hence, real-user results might be influenced depending on the mobile data providers and devices actually used.

## Performance score

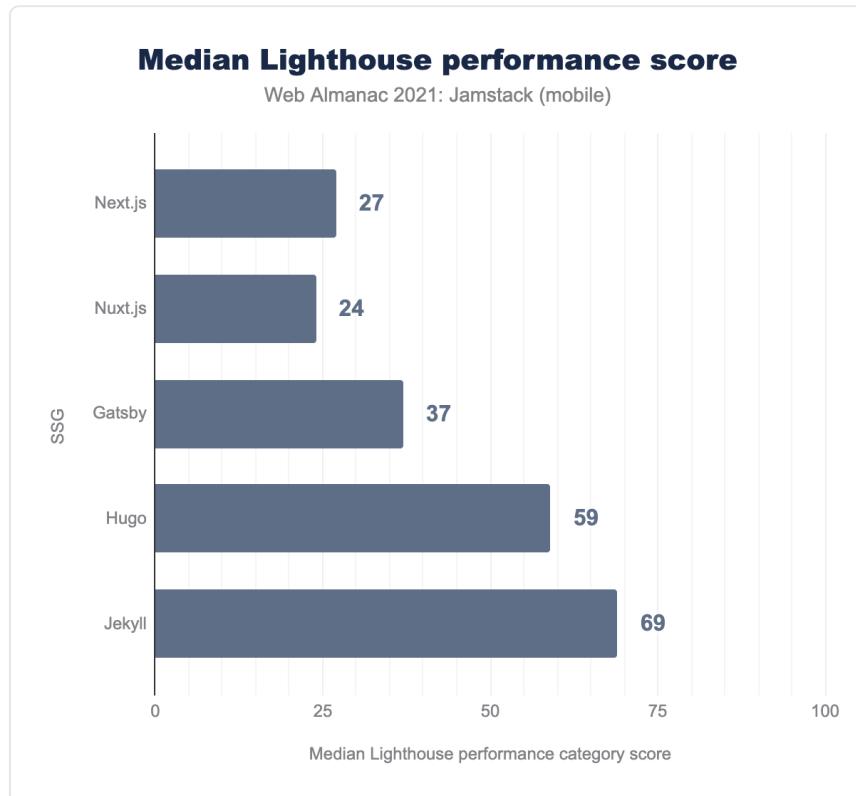


Figure 18.10. Median Lighthouse performance score.

The median performance score for all SSGs across mobile varies. The top 3 SSGs with by popularity can't even surpass a score of 40. Since they are used in top ranking sites and since users are likely distributed all around the world, we can assume that they are used across many different devices and networks. We can expect more out-of-the-box improvements like Next.js image component<sup>868</sup> to help performance.

Jekyll is a stand out, achieving a score of almost 70 which is a great result for such a mastodon in the SSG area. Learn more about Lighthouse performance audit<sup>869</sup> to understand exactly what measures are included in this score.

868. <https://nextjs.org/docs/basic-features/image-optimization>

869. <https://web.dev/lighthouse-performance/>

## Accessibility score

Lighthouse also runs audits to measure accessibility<sup>870</sup> and here we seem to have better results:

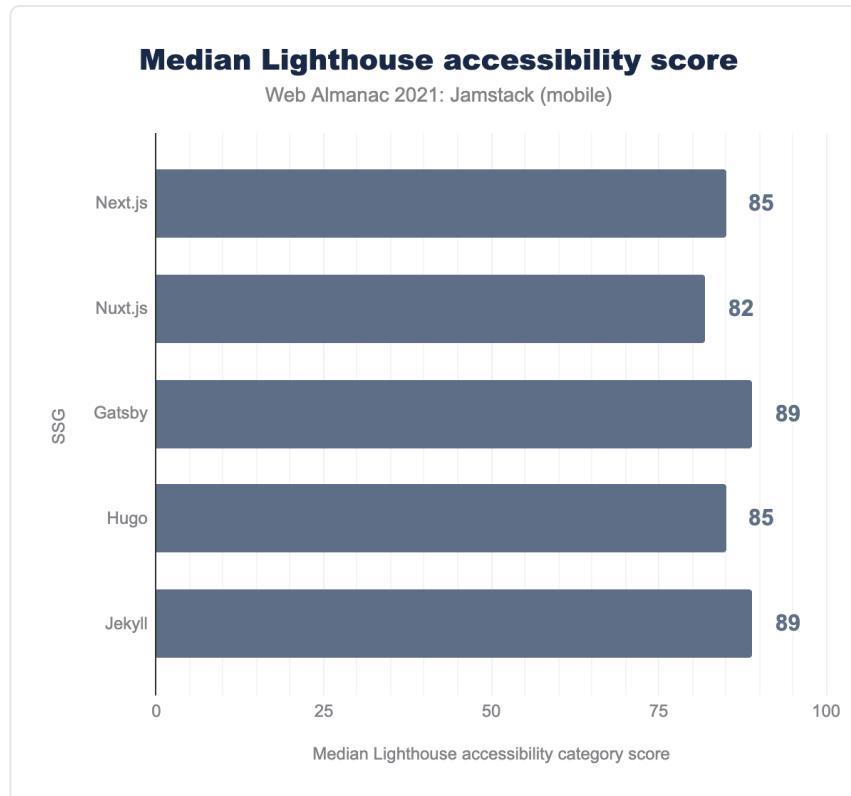


Figure 18.11. Median Lighthouse accessibility score.

There are limits to what can be checked in an automated accessibility check, but this is still a positive sign. Read the Accessibility chapter for more on this subject.

870. <https://web.dev/lighthouse-accessibility/>

**SEO score**

Figure 18.12. Median Lighthouse SEO score.

Similarly, all Jamstack sites provide great SEO scores from 90 to 92. Using static content always was SEO-friendly technique by default. Moreover, SSGs allow additional out of the box functionality to optimize sites for search engines.

The bottom line here is that Lighthouse results in general are good, but performance and PWA should be the main target for SSGs, these categories need some work to improve developer experience out of the box, hence the end result of sites performance will be improved.

## Core Web Vitals

Core Web Vitals<sup>871</sup> (CWV) is an initiative to provide unified guidance for quality signals that are essential to delivering a great user experience on the web. CWV itself uses 3 performance metrics:

- Largest Contentful Paint (LCP) - which measures the load time of the presumed main content of the page.
- First Input Delay (FID) - which measure interaction delays.
- Cumulative Layout Shift (CLS) - which measures visual stability so content is not moving around as the page loads and the user reads the content.

We used the *Chrome UX Experience Report* (CrUX) which gathers real-user data of these values and so is a better measure of actual user experience than the lab-based performance metric that Lighthouse provides.

We analyzed data for the SSGs, but this also reflects how those are delivered. As we saw above some sites are used more or less on different CDNs which may have a better (or worse!) impact on performance because of that so we also look at that data.

In the overall assessment for SSGs we can understand the basic performance level of Jamstack sites. CWV assessment contains data of 75% percentile of page loads which have a good score of CWV across all metrics.

---

<sup>871</sup> <https://web.dev/learn-web-vitals/>



Figure 18.13. Real-user Core Web Vitals compliance.

Looking at mobile results, Jekyll and Hugo have the best results over SSGs—34.3% and 31.8% of all sites scored good. Gatsby is third with 21.9%, but it's the first of the JavaScript-based SSGs. Next.js with 13.6% of good performance pages and Nuxt.js has 11.0%.

## Largest Contentful Paint

The Largest Contentful Paint<sup>872</sup> (LCP) metric reports the render time of the largest image or text block visible within the viewport, relative to when the page first started loading.

872. <https://web.dev/lcp/>



Figure 18.14. Real-user Core Web Vitals LCP.

Above we see the same results are approved by percent of sites with good LCP experience. The best results show Jekyll and Hugo with 76.4% and 70.3% of mobile sites having a “good” LCP of under 2.5s. The JavaScript based SSGs (Gatsby, Next.js, and Nuxt.js) fair worse.

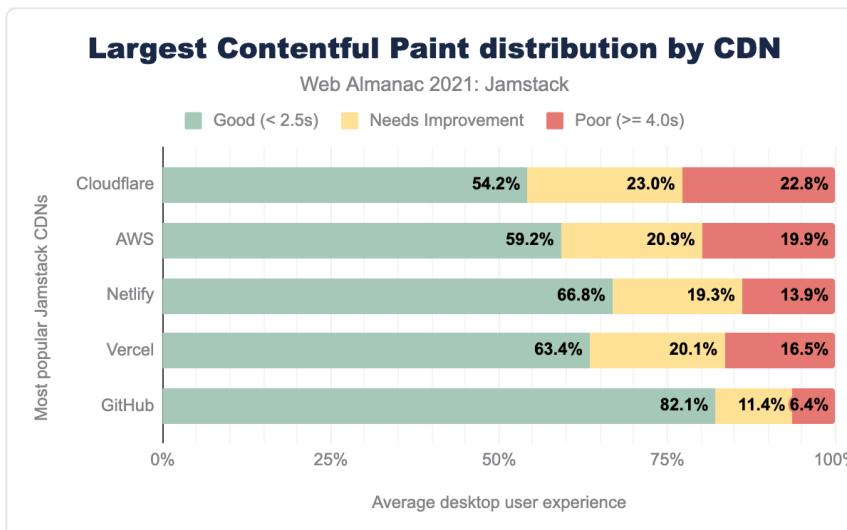


Figure 18.15. LCP distribution for CDNs.

GitHub tops the stats when measuring on CDN level, likely reflecting the simpler sites hosted here. Netlify, a Jamstack-oriented CDN, comes next with 66.8% of sites having a good LCP followed by Vercel with 63.4% followed by AWS with 59.2% and Cloudflare at 54.2%.

## First Input Delay

First Input Delay<sup>873</sup> (FID) measures the time from when a user first interacts with a page (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 begin processing event handlers in response to that interaction.

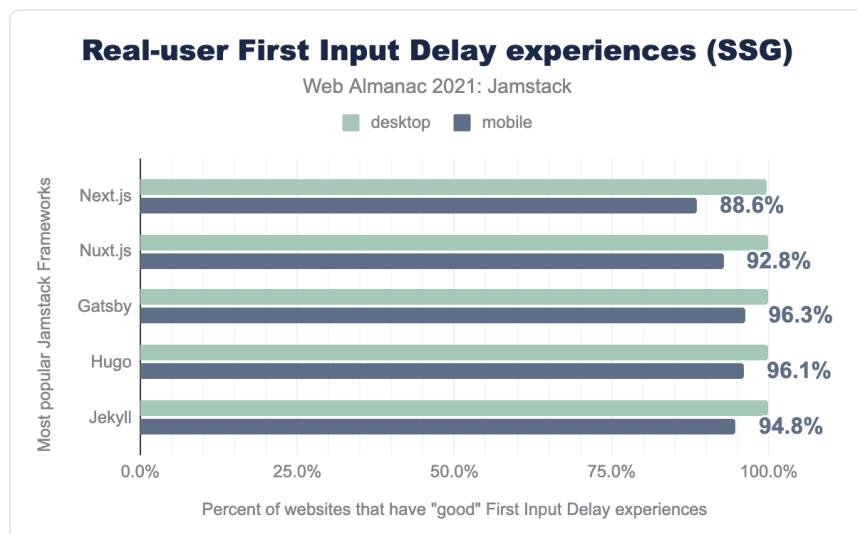


Figure 18.16. Real-user Core Web Vitals FID.

On a real user experience, All SSG show great FID results across different SSGs.

<sup>873</sup> <https://web.dev/fid/>

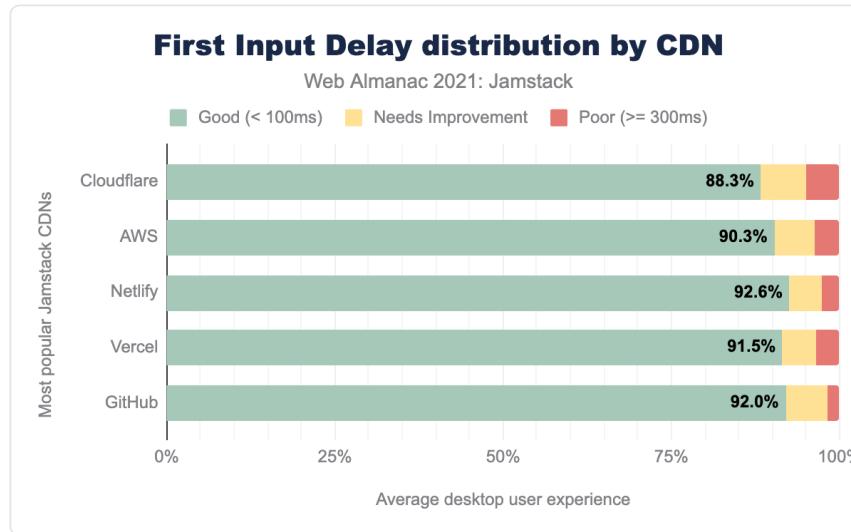


Figure 18.17. FID distribution for CDNs.

All CDNs deliver Jamstack sites with 88% good FID or above, though interesting that the Cloudflare and AWS sites fare slightly worse than the Jamstack-orientated CDNs.

## Cumulative Layout Shift

Cumulative Layout Shift<sup>874</sup> (CLS) is a measure of the largest burst of *layout shift scores* for every unexpected layout shift that occurs during the entire lifespan of a page.

874. <https://web.dev/cls/>

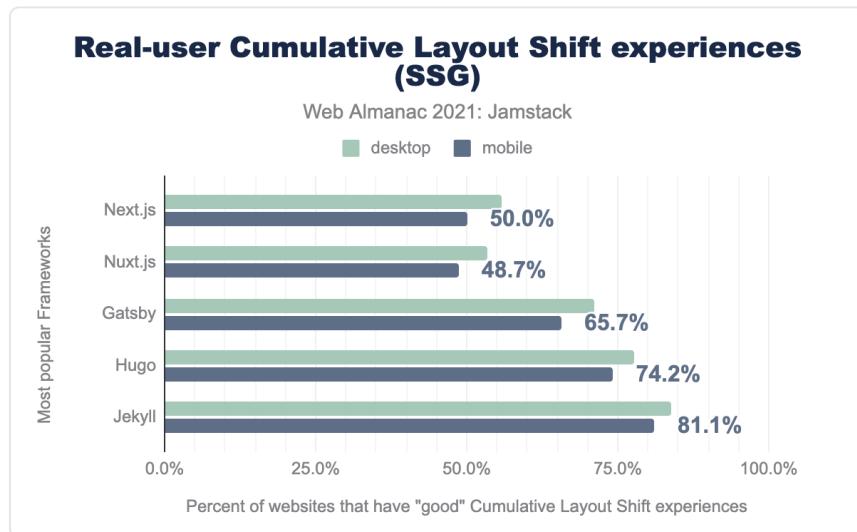


Figure 18.18. Real-user Core Web Vitals CLS.

Again, Jekyll shows great performance here. 81.1% of mobile are good results. Followed by Hugo at 74.2%, Gatsby at 65.7%, Next.js at 50.0%, and Nuxt.js trailing the pack at 48.7%.

Here's the same results as with previously for CDNs. GitHub, Netlify, Vercel.

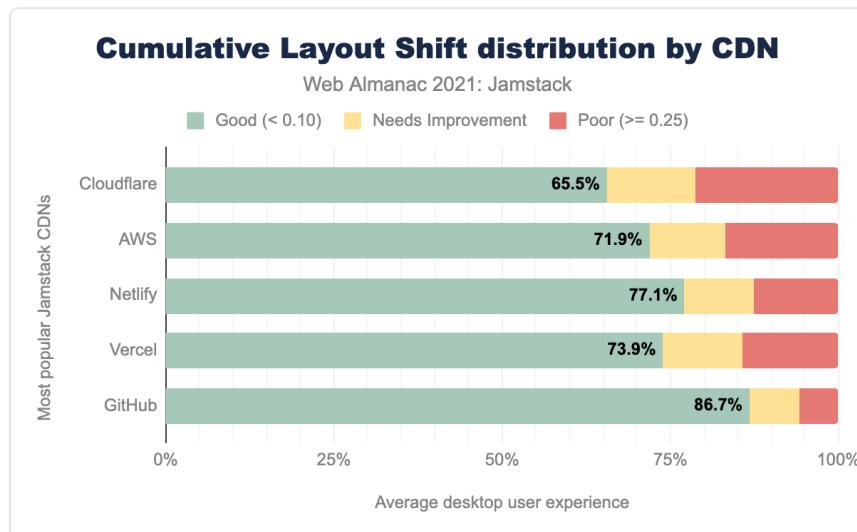


Figure 18.19. CLS distribution for CDNs.

In general CWV results reflect Lighthouse results. Huge and Jekyll have better real user performance data. We can't detect how complicated sites were built with these SSGs. We can bet that with modern SSGs like Next.js, Nuxt.js, Gatsby there are a lot of JavaScript delivered, more data to render including images. Hence, it affects performance results. Nevertheless, an interesting correlation between GitHub and Jekyll, which in tandem shows great results.

## Resources

Let's dive into resource weights between top fives SSGs to understand their influence on performance. The results represent median values.

### Resources weight

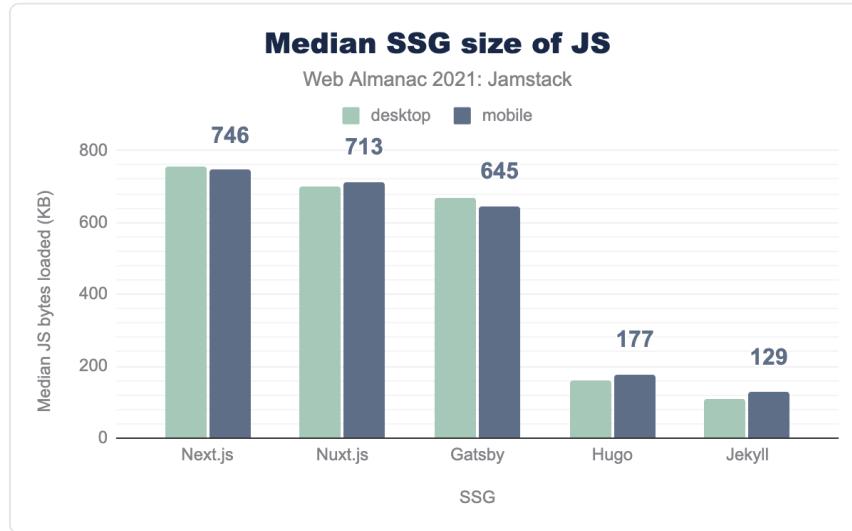


Figure 18.20. Median page weight.

JavaScript based SSGs have almost 2 times larger amount of resources than Hugo and Jekyll. The top one is ~2 MB for Nuxt.js, followed by Next.js and Gatsby with almost 1.8 MB and 1.7 MB.

As we mentioned above, JavaScript-based SSGs include JavaScript frameworks out of the box. That makes development easier, but requires more responsibility. The JavaScript ecosystem makes it easy to add more and more libraries to a site, for various purposes, which can lead to large bundle sizes.

## JavaScript



*Figure 18.21. Median JavaScript weight.*

A big chunk of resources are for JavaScript. Again, for JavaScript-based SSGs it's a much bigger compared to others - around 700 KB compared to around 150 KB for non-JavaScript based SSGs. While this is not surprising, it's interesting to see the actual differences laid out in this way. Next.js based sites use more JavaScript than others. Hugo and Jekyll developers on other hand seem to be using JavaScript more responsibly and keeping their bundles tight. Another reason for that might be site complexity. Hugo and Jekyll sites are not represented as much in top ranking sites, so they might have simpler use cases than, for example, Next.js sites which do appear more often in the top ranking sites.

We analyzed which third party libraries were used among SSGs. We excluded React and Vue to have a clear picture of other libraries and frameworks represented among SSGs.

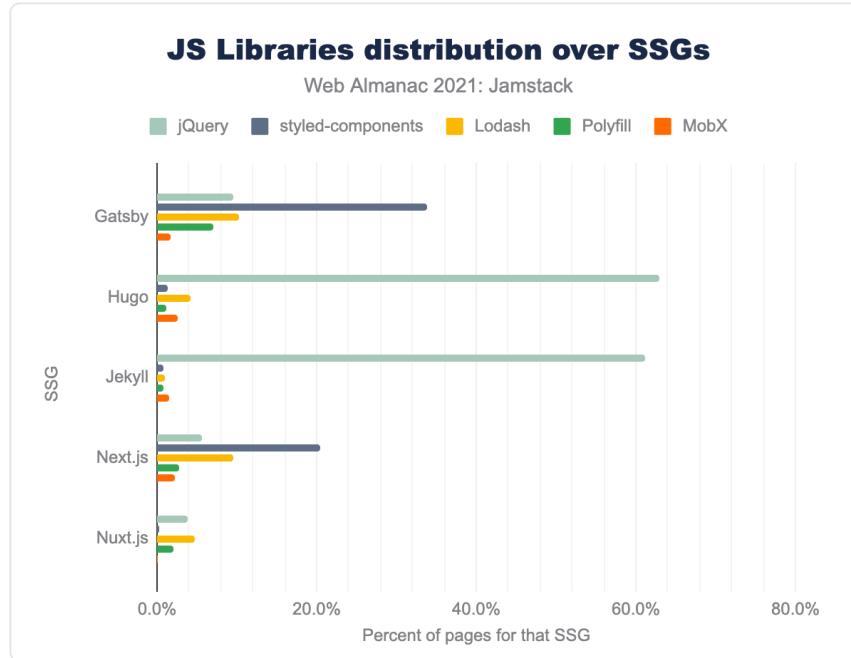


Figure 18.22. JavaScript 3rd parties distribution over SSGs.

A big surprise for us was jQuery. It wasn't a surprise that it's used for Hugo and Jekyll based sites (more than 60%), but that it's used inside React and Vue based sites wasn't expected! Next.js, Many Nuxt.js, and Gatsby sites use jQuery too.

Styled-components was used for Next.js - 20% and Gatsby takes 34% from all of third party libraries. Nuxt.js sites almost don't use it.

Lodash is heavily used and was present among all SSGs up to 10% for Gatsby.

## CSS

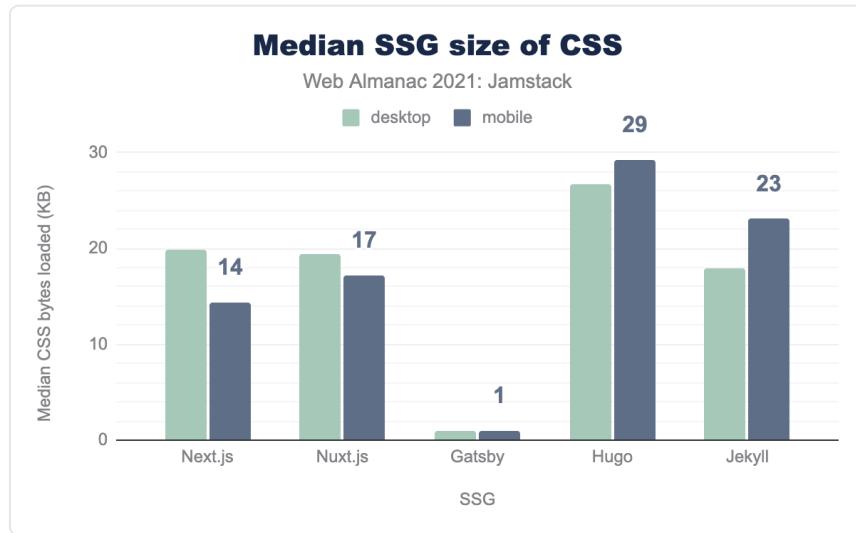


Figure 18.23. Median CSS weight.

On the other hand, CSS is slightly heavier than Hugo and Jekyll. Since one of the benefits of styled-components is clean, non-repetitive CSS, this could explain why CSS size for these JavaScript SSGs are lower. One more hypothesis is that old fashioned SSGs use old fashion methods for handling interactions and animations using CSS. JavaScript-based SSGs use more JavaScript in general, hence they might more often be used to replace functionality that could be implemented with CSS.

## Images

Images weights distributed differently. There's no correlation between SSG groups.



Figure 18.24. Median image weight.

Nuxt.js has the highest value at 645 KB. Hugo is next with 522 KB. Next.js and Gatsby are almost the same at 465 KB and 454 KB respectively. Jekyll has the lowest value at 295 KB.

## Images format adoption

Images are one of the bottlenecks of good User Experience (UX). If they are large, then the user has to wait for a long time for the image to be delivered. It can lead to layout shifts and other problems.

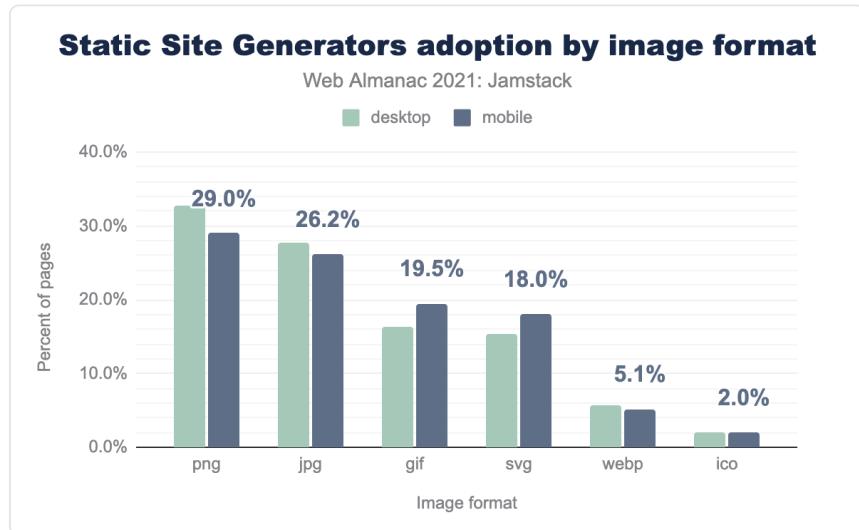


Figure 18.25. Adoption of image format.

As one of the newer generation of image formats, WebP<sup>875</sup> has 5.1% of usage among Jamstack sites. Compared to last year's results<sup>876</sup>, when WebP had only 3%, we can say it's a good improvement over one year.

Still, the most used is PNG at 29.0% and JPEG at 26.2%, GIF at 19.5%, and SVG is used on 18.0% of web pages.

## What the resources tells us

This analysis of resource weights confirms that performance of Next.js, Nuxt.js and Gatsby are likely struggling because of huge resources. 2 MB of page weight and ~ 700KB of JavaScript that will definitely have an impact on performance scores, especially for average mobile devices and slower networks. Heavy usage of styled-components for Next.js and Gatsby sites might be another cause of lesser performance<sup>877</sup>. A positive signal is that image adoption of next-generation image formats is growing, and this should improve UX for end users in the long run.

## Conclusion

Despite limitations on not being able to include headless CMSs, and for some well-known SSGs

875. <https://developers.google.com/speed/webp/>

876. <https://almanac.httparchive.org/en/2020/jamstack#image-formats>

877. <https://pustello.com/blog/css-vs-css-in-js-perf/>

(Eleventy or Next.js detection mode), we still have a lot of data to analyze here to draw some interesting conclusions. The Jamstack trend is growing year over year: now more than 1% of all websites are Jamstack based.

We know that Next.js covers around 40% of measurable Jamstack sites. It's not only trending, but also used in 3.8% of the top 1,000 sites followed by the other popular SSGs such as Nuxt.js and Gatsby. These are all relatively new players just a few years in the space but they have solidified their place by good usage among top ranked sites as well.

SSGs are used all around the world, and are not confined to those countries with the founding companies of this model are based. In fact it seems that some of the fastest-growing adopters of Jamstack technology, with up to 5% of sites, are those regions furthest away from the tech hubs of Silicon Valley.

Like all websites, maintaining good performance of Jamstack sites requires knowledge of best practices and experienced developer level to achieve good results, but SSGs can improve this by working on out-of-the-box solutions to improve in that area. Hope you enjoyed the data and give Jamstack a try.

## Author

---



Artem Denysov

 @denar90\_  denar90

Artem Denysov is Software Engineer, Open-Source contributor, proud Mozillians member, speaker, and writer. Makes developers and users live easier helping them with webperf & tools. Works at Stackbit<sup>878</sup> to empower developers build Jamstack websites easily. You can find him on Twitter<sup>879</sup> and LinkedIn<sup>880</sup>.

---

878. <https://stackbit.com>

879. <https://twitter.com/denar90>

880. <https://www.linkedin.com/in/denar90/>

# Part IV Chapter 19

# Page Weight



**Written by John Teague**

*Reviewed by Sia Karamalegos and Rebecca Holmlund*

*Analyzed by Jess Peck*

*Edited by Barry Pollard*

## Introduction

Unless you're a web performance junkie like me, the weight of a web page is about as exciting as licking stamps. But, I'm going to try my best to convince you as to why page weight is not only important but arguably *the* most important factor affecting creators, hosting providers, and consumers. To that end, we'll use real data to show how the weight of a page influences the performance of the website or web application, how page weight can impact user experience, and some ways we can reduce the weight of our web pages.

In the past decade, average web page weight<sup>881</sup> has grown a whopping 356 percent, from an average of about 484 kilobytes to 2,205 kilobytes. That increase can be explained as a function of supply and demand. Faster computer processors, data transmission, and how data is stored and made available have all advanced to keep up with increased use of images, video, audio, fonts, data collection and processing, and connected services like analytics, monitoring, and

881. <https://httparchive.org/reports/page-weight>

alerting functionality for web sites and web applications.

All seems well, if you're fortunate enough to own a high end smartphone, desktop or laptop computer costing thousands of dollars, and you're connected to an expensive high speed internet provider or 5G data plan. But the pleasure of belonging to that class of internet user starts to break down when you're relegated to using a slow 3G or 4G data plan with unpredictable internet connectivity. For a large segment of internet users, waiting for a page that may never fully load breaks the promise of the internet even to the point of putting lives at risk during emergencies<sup>882</sup>.

A lot of energy is used to power data centers and the devices they serve. We can help reduce overall energy demands by keeping our file payloads smaller which also keeps payload transmission faster and more efficient.

Google now penalizes a website's search ranking for those that fail to achieve good Core Web Vitals. One of their metrics for assessing success or failure is page weight. If you are interested, you can test your site using Google PageSpeed Insights<sup>883</sup> and Google Measure<sup>884</sup>. Both provide valuable insights into how to solve performance and user experience problems caused by heavy web pages.

To understand and find opportunities to keep web pages lighter and faster, it's instructive to examine what page weight actually is. So let's delve deeper.

## What is page weight?

Page weight describes the total number of bytes of a particular web page. A web page is comprised of specific elements and assets that can be rendered and viewed in a web browser, including:

- The HTML that makes up the page itself.
- Images and other media (video, audio, etc) embedded into the page.
- Cascading Style Sheets (CSS) used for styling the page.
- JavaScript to provide interactivity
- Third-Party resource containing one or more of the above.

Each of those resources exact a cost in weight (byte size), and computational resources to

---

882. <https://www.nbcnews.com/tech/tech-news/verizon-admits-throttling-data-calif-firefighters-amid-blaze-n902991>

883. <https://pagespeed.web.dev/>

884. <https://web.dev/measure/>

transmit, process and render in a web browser. While they have similar cost in some regards (storage and transmission), the CPU cost of some resource types may be more costly in those regards than others.

The process of managing web page resources for use when requested have rapidly changed over the past decades. Part of those changes were predicated on making web page resources more efficient and more quickly transmittable when requested. Let's examine three impacts of page weight for resources:

## Storage

Page resources need to be stored ready for retrieval when requested. Image, video, CSS, JavaScript, and font files assets are stored in multiple places: on servers, on local devices, and in memory. Each file, ranging from a few bytes to many megabytes in size, therefore has a cost impact in multiple places. While server storage costs may seem relatively cheap, limited storage on devices can result in assets being evicted from caches or memory resulting in more downloads and more costs.

Many people don't understand, or pay little attention to, the negative impact those types of unoptimized assets have on page loading performance. When reviewing today's websites, I routinely discover images that exceed four megabytes in size, and embedded video files that are many times that value.

Fortunately, there are also options and optimizations that can be applied that can significantly lower the size of files stored at rest from compression, to using the appropriate file format for media to offloading content to a dedicated CDN who can handle this for you to lighten the weight of a web page, often at little to no cost.

## Transmission

When a user requests a web page via HTTP, all files needed by the page are then requested. Files are located and sent back to the requesting device and, if all goes well, the requester's browser will take the payload, and process and render it as part of the larger web page on the requesting user's screen. Page weight becomes important during the transmission process because the size of the file determines how long it will take to complete the transfer of the resources, which will then ultimately impact the rendering of the results.

A negative effect of large page weight is due to *latency* and *bandwidth* constraints. Latency measures the time it takes for the request to connect to the server storing the files and begin the process of transporting those files, while bandwidth measures the time it takes to download the resources. If a bunch of files are requested, no matter the technology, there is a limit on how

much can be processed and transferred in any given period. I've audited WordPress sites that request as many as 170 files or more, which ensures terrible page loading performance starting with high latency periods.

Many optimizations can improve transfer/loading time, such as compressing and combining certain file requests, using HTTP/2—or the newer HTTP/3—protocols, and using a modern browser's ability to preconnect to and preload certain files to speed the whole process up, but ultimately page weight will still have an impact here. The Performance chapter covers a wide range of factors that effect page loading performance.

## Rendering

A web browser is ultimately software that makes requests to for resources on behalf of users (hence the term *user agent*). The results of those requests are handed off to the browser's rendering engine to process and then recreate the web page you asked for. It's not hard to deduce that the larger the total amount of page weight, the more the browser engine must process and render to the browser screen, and so the longer it's going to take.

If too many files, especially large media and large complex scripts must get retrieved, read, processed, and then finally rendered by the browser before the content becomes available, then this increase the chance that pages will take so long to load that users will abandon them.

Large payloads can also overwhelm the amount of client-side resources available on the user's smartphone or computer causing it to stall and even crash the device. Users who have the good fortune to subscribe to high speed cable internet services, or 5G data plans for high end devices will seldom experience these problems. But again, a large percentage of internet users don't have access to those levels of internet services and devices.

## Assets

As explained in last years chapter<sup>885</sup>, we have not really changed what types of assets are used on web pages over the years, but there are some notable exceptions.

## Images

Static files reside by themselves and are used as resources to help build out and render web pages. Images, video, audio, and font files are all examples of static assets. Images make a large percentage of the average web page's weight so, let's use images for our example.

---

<sup>885.</sup> <https://almanac.httparchive.org/en/2020/page-weight#assets>

Image formats like PNG and JPEG are widely supported by all browsers. More recent image formats, such as WebP and AVIF offer higher quality with smaller file sizes have gained popularity. WebP is supported by most modern browsers, while AVIF is newer and less supported. With the `<picture>` tag, you can use modern image formats while providing JPEG and PNG fallbacks. Make sure your images are optimized for the web—the Media chapter covers this in much more detail. Failing to properly size and compress images for your site will exact a high price on performance.

*Note: If you need an online service that will optimize and allow you to compare different image sizes formats, there is no better source I've found than Google's Squoosh<sup>886</sup> application. Similarly, Jake Archibald<sup>887</sup>'s SVGOMG<sup>888</sup> is great for optimizing SVG's.*

## A word about the proliferation in the use of JavaScript

JavaScript can be a wonderful tool to use for creating a dynamic website, but using it unchecked can create serious performance problems and a horrible experience for the user. There's been a proliferation in the use of complex JavaScript web frameworks and libraries over the past decades and the sheer amount of JavaScript is a large percentage of total page weight. Some JavaScript can cause sizes for a site to skyrocket leading to serious performance bottlenecks. Some are so bad that a site can become unstable or even unusable. Blocking scripts, that must be transmitted, processed, and executed before the page can finish rendering enough page assets for users to interact with it. That can cause confusion, frustration, and abandonment by the user.

Nine times out of ten when a site stalls, it is a blocking JavaScript that is causing your smartphone to run out of processing resources or memory is to blame. The judicious, expert use of JavaScript can create great user experiences. But remember this: JavaScript is executed on the client side. It's using the client computers resources to process and execute the script, and there is a finite amount of resources on every device. Once again, not everyone is glued to the newest Google Pixel or Apple smartphone. The JavaScript chapter contains a wealth of information about this issue.

## Third-party services

Page weight can also be affected by external services called by web page. Some of those services include CDN's, analytics, chat bots, forms, and other data collection and processing methods. I find this to be one of the fastest growing problem areas that result in bloated page weight. Many of these third-party services use outdated, poorly-written JavaScript and

---

886. <https://squoosh.app/>

887. <https://twitter.com/jaffathecake>

888. <https://jakearchibald.github.io/svgomg/>

querying techniques that take much longer to execute than they should, and the site owner has little control over how that third party impacts the loading of a page. Suffice it to say that inquiring about how a service will affect your page loading performance is very important. So is testing their impact.

## Caching

Caches, are allow resources to be served quickly, thus avoiding the cost of the download again. Caches exist on both users' browser, but also on servers. Caching of optimized assets dramatically lowers page weight and page loading time because the asset is immediately available, removing the need to execute an entire request process. While not reducing the overall page weight, they can help reduce the impact.

## Page weight by the numbers

Looking at the page weight on both desktop and mobile devices, the difference is generally small between them despite the often-different capabilities of these devices:



*Figure 19.1. Distribution of total bytes per page.*

We are closing in on 6.9 MB of page weight on mobile and 8.1 MB on desktop at the 90th percentile.



Figure 19.2. Median bytes per page by content type.

A closer inspection at the median, shows that the images remain the largest resource followed by JavaScript.

Let's look at the growth over time:



Figure 19.3. Median page weight over time.

The trend of page weight growth couldn't be clearer. We're on an upward trajectory that shows no sign of abating.

## Requests

As previously explained in this chapter, as well as the size of resource, the number of requests can have negative impact on page loading performance and so are another measure of page weight.

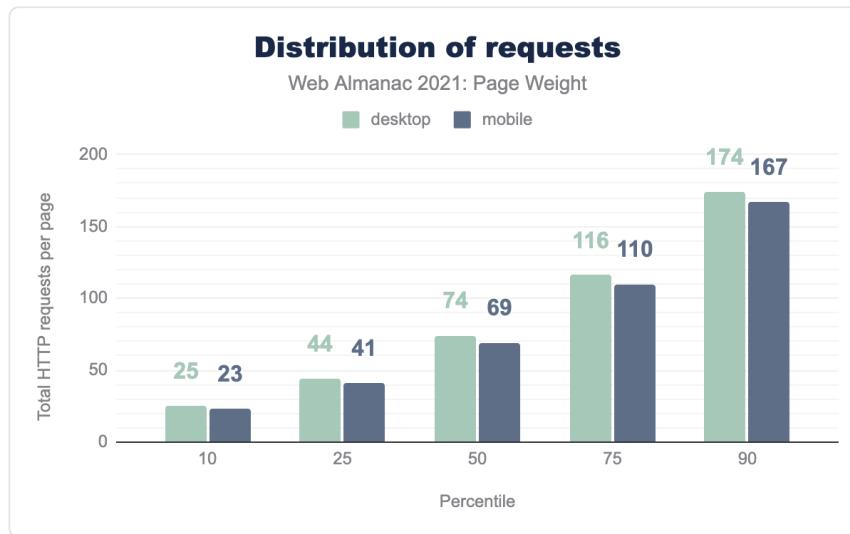


Figure 19.4. Distribution of requests per page.

The request distribution shows that the difference between desktop and mobile is not significant, with desktop leading the way.

The difference between current results for this year and last actually shows a tiny decrease in the average number of GET requests across most of the percentiles. Let's hope that trend continues downward.

Something else worth noting: the median request on desktop at this time is the same as last year<sup>889</sup> (74), yet the page weight has ticked up (141 kb).

<sup>889</sup>. <https://almanac.httparchive.org/en/2020/page-weight#requests>

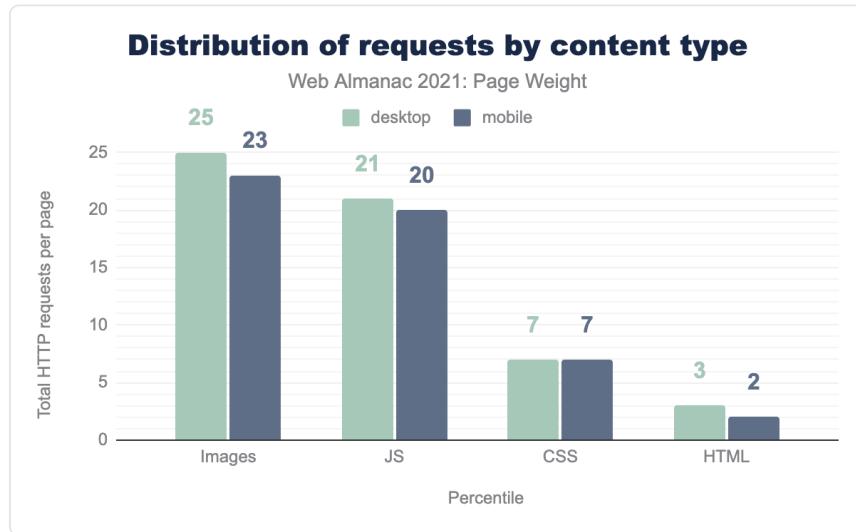


Figure 19.5. Median number of requests by content type.

Images again make up the largest number of requests, though JavaScript is closing in as the gap has narrowed slightly in the last year. Images shows a reduction of 4 requests between the two years—perhaps a result of more lazy-loading<sup>890</sup> since this was made available natively via simple HTML attributes?

890. [https://developer.mozilla.org/docs/Web/Performance/Lazy\\_loading](https://developer.mozilla.org/docs/Web/Performance/Lazy_loading)

## File formats

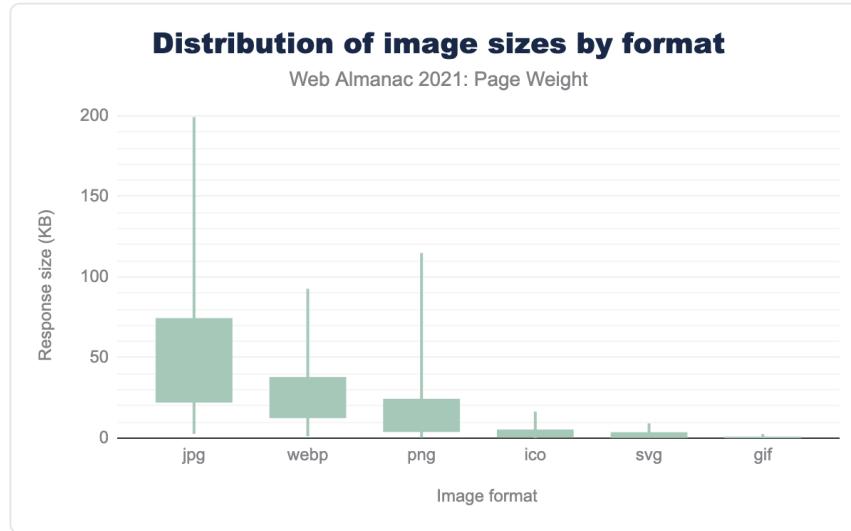


Figure 19.6. Distribution of image sizes by format.

We know images are responsible for a large percentage of web page weight. The above graphic shows the top sources of image weight and the weight distribution. Top 3: JPG, WebP and PNG. Compared to last year, we see an increase in WebP usage now it is finally supported in all major browsers. PNG remains popular for use cases such as icons and logos.

## Image bytes



Figure 19.7. Distribution of image response sizes per page.

Looking at total image bytes shows us that this metric has remained virtually unchanged from the previous year<sup>891</sup>. One reason for this could be an increase in the number images being served by content distribution networks (CDN), which apply strong optimizations to images as they are uploaded to their servers thus keeping any growth in check for new images.

## Conclusion

How important is it to keep web pages light? Overall page weight affects page loading speed, and page loading speed affects user experience. Google's Web Vitals program focuses on user experience, especially for mobile users, with a direct impact on Google Search rankings. So, there is a real incentive and a real consequence to keep web pages as light as possible.

But will impact on search rankings translate into direct pressure to lighten page loads? What about web titans, like Amazon? Is there incentive for hugely popular web sites to worry about page weight? Perhaps. The Amazon's may want to take advantage of reducing the size of page assets and services to reduce the spend required to serve those pages, or maybe they want to move into newly emerging markets where users may not be able to buy super-fast smartphones or have access to 5G data networks or high-speed cable providers. Time will tell.

<sup>891</sup> <https://almanac.httparchive.org/en/2020/page-weight#file-formats>

## Author

---



John Teague

Twitter: [@jtteag](https://twitter.com/jtteag) GitHub: [logicalphase](https://github.com/logicalphase) Website: <https://gemservers.com>

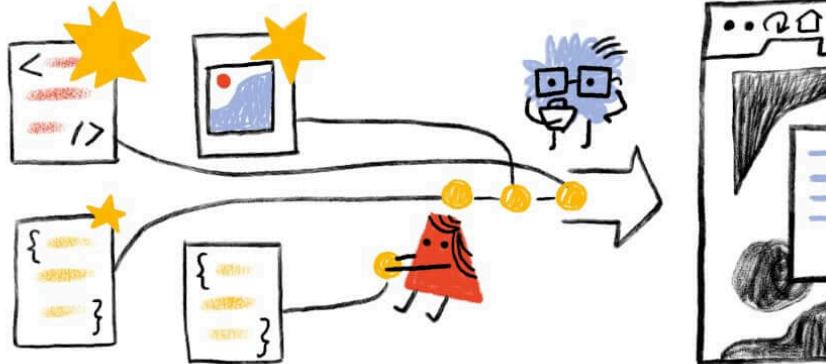
John currently works as a Google Cloud Platform<sup>892</sup> senior developer and architect. He started his technology journey as a web developer focused on web performance and leveraging browser standards. He applied those principles as a freelance WordPress<sup>893</sup> developer, and as an architect and engineer for several managed hosting providers. He is a firm believer in open web standards and sustainable web best practices. To that end, John has worked on several open source projects, including Google's Lit<sup>894</sup> project, and is a strong advocate for emerging web technologies such as Web Components<sup>895</sup> and other performance based solutions.

---

892. <https://cloud.google.com>  
893. <https://wordpress.org>  
894. <https://lit.dev/>  
895. [https://developer.mozilla.org/docs/Web/Web\\_Components](https://developer.mozilla.org/docs/Web/Web_Components)

# Part IV Chapter 20

# Resource Hints



Written by Kevin Farrugia

Reviewed by Sia Karamalegos, Barry Pollard, Andy Davies, Samar Panda, and Weston Ruter

Analyzed by Nitin Pasumarty

Edited by Rick Viscomi

## Introduction

Resource hints are instructions to the browser that you may use to improve a website's performance. This set of instructions enable you to assist the browser in prioritizing origins or resources which need to be fetched and processed.

Let's take a closer look at how resource hints are implemented, what are the most common pitfalls, and what we can do to make sure we are using resource hints as effectively as possible.

### The Link directive

The most widely adopted resource hints are implemented through the `Link` directive's `rel` attribute. These are `dns-prefetch`, `preconnect`, `prefetch`, `prerender` and `preload`.

These may be implemented in one of two ways:

## HTML element

```
<link rel="dns-prefetch" href="https://example.com">
```

## HTTP header

```
Link: <https://example.com>; rel=dns-prefetch
```

It is also possible to dynamically inject the HTML element through the use of JavaScript:

```
const link = document.createElement("link");
link.rel="prefetch";
link.href="https://example.com";
document.head.appendChild(link);
```

Adoption for HTTP headers is significantly lower than having resource hints implemented as part of the document markup; with less than 1.5% of the pages analyzed implementing resource hints through HTTP headers. This is likely attributed to the ease with which they may be added or modified from within the HTML source, when compared to adding an HTTP header on the server.



Figure 20.1. Popularity of resource hints as HTTP headers and HTML markup.

Using our current methodology, it is not possible to reliably measure resource hints that are added following user-interaction, such as those added through QuickLink<sup>896</sup>, though that particular library featured on less than 0.1% of pages analyzed, according to the Core Web Vitals Technology Report<sup>897</sup>.

Considering that the adoption of resource hints using HTTP headers is markedly smaller than adoption for the `<link>` HTML element, the rest of this chapter will focus on analyzing the usage of resource hints through the HTML element.

## Types of resource hints

There are five resource hint link relationships supported by most browsers today: `dns-prefetch`, `preconnect`, `prefetch`, `prerender` and `preload`.

### `dns-prefetch`

```
<link rel="dns-prefetch" href="https://example.com/">
```

<sup>896.</sup> <https://github.com/GoogleChromeLabs/quicklink>

<sup>897.</sup> <https://datastudio.google.com/s/uMbv5CQfW4Q>

The `dns-prefetch` hint initiates an early request to resolve a domain name. It is only effective for DNS lookups on cross-origin domains and may be paired together with `preconnect`. While Chrome now supports a maximum of 64<sup>898</sup> concurrent in-flight DNS requests—up from 6 last year—other browsers still have tighter limitations. For example, it is limited to 8<sup>899</sup> on Firefox.

### preconnect

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

The `preconnect` hint behaves similarly to `dns-prefetch`, but in addition to DNS lookups, it also establishes a connection together with TLS handshake if served over HTTPS. You are able to use `preconnect` in place of `dns-prefetch` as it gives a greater performance boost; but you must use it sparingly as certificates are usually upwards of 3 KB, which would be competing with bandwidth for other resources. You also want to avoid wasting CPU time opening connections which aren't required for critical resource. Keep in mind that if a connection isn't used within a short period of time (e.g., 10 seconds on Chrome), it would automatically be closed by the browser, wasting any `preconnect` effort.

### prefetch

```
<link rel="prefetch" href="/library.js" as="script">
```

The `prefetch` hint allows you to recommend to the browser that a resource might be required by the next navigation. The browser may initiate a low-priority request for the resource, possibly improving the user experience as it would be fetched from the cache when needed. While resource may be fetched in advanced with `prefetch`, it will not be preprocessed or executed until the user navigates to the page which requires the resource.

### prerender

```
<link rel="prerender" href="https://example.com/page-2/">
```

898. [https://source.chromium.org/chromium/chromium/src+/fd9418d23d434e0f7134da67dc41b0fe8268e91:net/dns/host\\_resolver\\_manager.cc;l=416](https://source.chromium.org/chromium/chromium/src+/fd9418d23d434e0f7134da67dc41b0fe8268e91:net/dns/host_resolver_manager.cc;l=416)  
899. <https://github.com/mozilla/gecko-dev/blob/master/netwerk/dns/nsHostResolver.h#L48>

The `prerender` hint allows you render a page in the background, improving its load time if the user navigates to it. In addition to requesting the resource, the browser may preprocess and fetch and execute subresources. `prerender` could end up wasteful if the user does not navigate to the prerendered page. Contrary to the specification, Chrome treats the `prerender` hint as a NoState Prefetch<sup>900</sup> to reduce this risk. Unlike a full prerender it won't execute JavaScript or render any part of the page in advance but only fetch the resources in advance.

## preload

Most modern browsers also support<sup>901</sup> the `preload` hint—and to a lesser degree<sup>902</sup>, the `modulepreload` hint. The `preload` instruction initiates an early fetch for a resource which is required in the loading of a page and is most commonly used for late-discovered resources, such as font files or images referenced in stylesheets. Preloading a resource may be used to elevate its priority, allowing the developer to prioritize the loading of the Largest Contentful Paint<sup>903</sup> (LCP) image for, even if this would otherwise be discovered while parsing the HTML.

`modulepreload` is a specialized alternative to `preload` and behaves similarly, however its usage is limited to module scripts<sup>904</sup>.

---

900. <https://developers.google.com/web/updates/2018/07/nostate-prefetch>  
901. <https://caniuse.com/link-rel-preload>  
902. <https://caniuse.com/link-rel-modulepreload>  
903. <https://web.dev/lcp>  
904. <https://html.spec.whatwg.org/multipage/webappapis.html#module-script>

## Adoption and trends

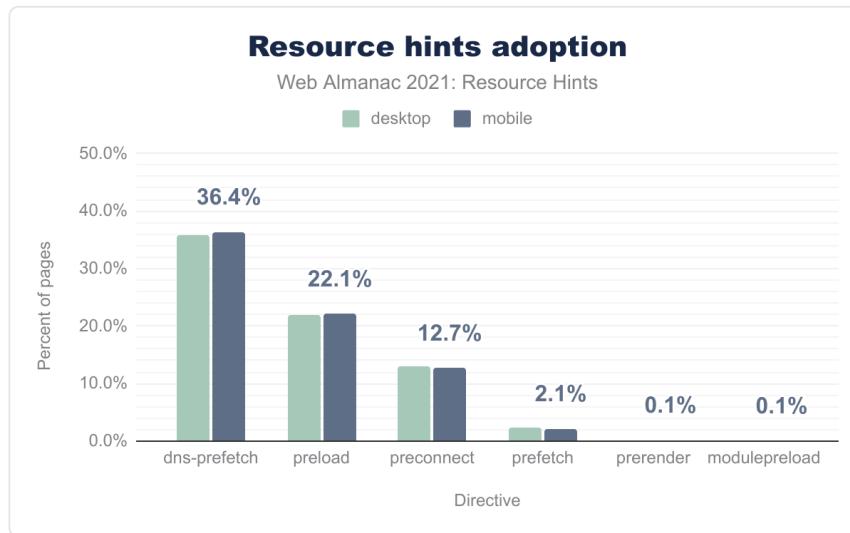


Figure 20.2. Adoption of the `link rel` attribute.

The most widely used resource hint is `dns-prefetch` (36.4% on mobile); which is unsurprising, considering it was introduced in 2009<sup>905</sup>. With the widespread use of HTTPS, in many cases you should replace it with `preconnect` (12.7% on mobile), if you are certain that you will be connecting to that domain. Considering that the `preload` hint is comparatively new, first appearing in Chrome in 2016<sup>906</sup>, it is the second most widely adopted resource hint (22.1% on mobile) and is seeing constant growth year-on-year—a testament to the importance and flexibility of this directive.

As shown in the charts above, the adoption rates on mobile and desktop are near-identical.

905. <https://caniuse.com/link-rel-dns-prefetch>  
 906. [https://groups.google.com/a/chromium.org/g/blink-dev/c\\_nu6HlbNQfo/m/XzaLnB1bBgAJ?pli=1](https://groups.google.com/a/chromium.org/g/blink-dev/c_nu6HlbNQfo/m/XzaLnB1bBgAJ?pli=1)

## By rank



Figure 20.3. Adoption of `rel="preload"` segmented by CrUX rank.

You can observe that when segmenting the data by rank, the adoption rates change notably, with the `preload` hint increasing from 22.1% for our whole data set, to claim the top spot with an adoption rate of 44.3% amongst the top 1,000 sites.

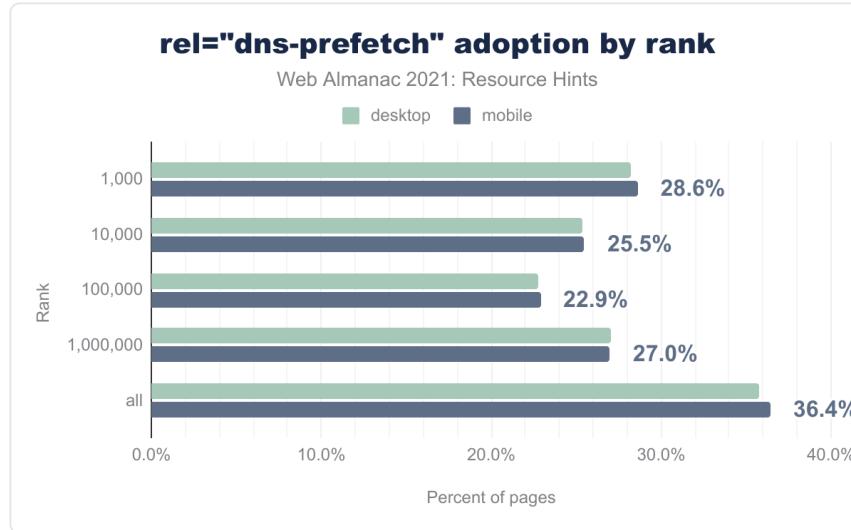


Figure 20.4. Adoption of `rel="dns-prefetch"` segmented by CrUX rank.

`dns-prefetch` is the only resource hint which exhibits a decrease in adoption when comparing the top 1,000 sites with the overall adoption.

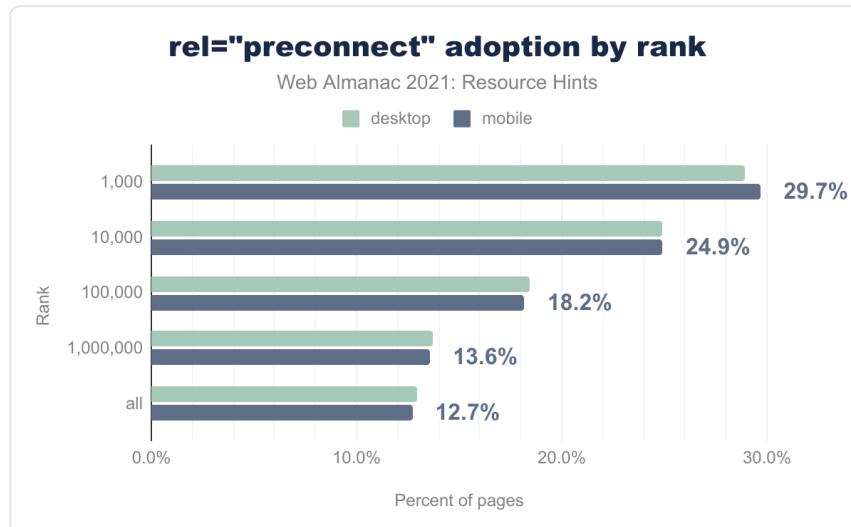


Figure 20.5. Adoption of `rel="preconnect"` segmented by CrUX rank.

To counter this decrease, the top 1,000 pages have an increased adoption for the `preconnect`

hint, taking advantage of its increased performance boost and wide support. I expect that the adoption for `preconnect` will continue increasing as the rest of the internet follow suit.

## Usage

Resource hints can be very effective if used correctly. By shifting the responsibility from the browser to the developer, it allows you to prioritize resources required for the critical rendering path and improve the load times & user experience.

<b>Rank</b>	<b>preload</b>	<b>prefetch</b>	<b>preconnect</b>	<b>prerender</b>	<b>dns-prefetch</b>	<b>modulepreload</b>	
1,000	3	2	4	0	4		1
10,000	3	1	4	1	3		1
100,000	2	2	3	1	3		1
1,000,000	2	2	2	1	2		1
<i>all</i>	2	2	1	1	2		1

Figure 20.6. Median number of resource hints per page by rank.

Of the sites using resource hints, when comparing the median for the top 1,000 sites to the entire corpus, the top-ranking sites have more resource hints per page. The only hint which observes a different pattern is `prerender`, which has a total of 0 occurrences in the top 1,000 sites.

## Correlation with Core Web Vitals



Figure 20.7. Correlation between good CWV score and number of `rel="preload"` hints

By combining a page's Core Web Vitals<sup>907</sup> scores in the CrUX dataset and the usage of the preload resource hint, you can observe a negative correlation between the number of link elements and the percentage of pages which score a good rating on CWV. The pages which use fewer preload hints are more likely to have a good rating.

907. <https://web.dev/vitals/>

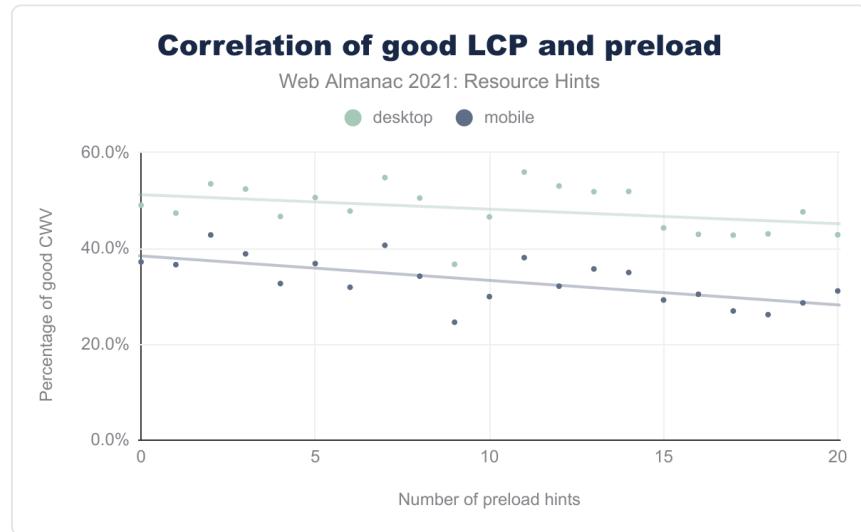


Figure 20.8. Correlation between good LCP score and number of `rel="preload"` hints

This same observation may be seen on a page's LCP, indicating that in many cases, the developer is prioritizing resources which aren't needed to render the LCP element and as a consequence degrading the user experience.

While this doesn't prove that having preload hints causes a page to get slower, having many hints does correlate with having slower performance. Every page has its unique requirements and it is impossible to apply a "one size fits all" approach, but in the majority of cases the number of preloaded resources should be kept low and resource prioritization should be delegated to the browser when possible.

*Note: In addition to the number of hints, the size of each preloaded resource has an impact on the website performance. The above figure does not take into consideration the size of each preloaded resource.*

## `rel="preload"`

With that being said, and the expectation that more websites will adopt `preload`, let's take a better look at the preload resource hint and understand why it is so effective, yet at the same time so prone to misuse.

## The `as` attribute

The `as` attribute should be specified when using `rel="preload"` (or `rel="prefetch"`) to specify the type of resource being downloaded. Applying the correct `as` attribute allows the browser to prioritize the resource more accurately. For example, `preload as="script"` will get a low or medium priority, while `preload as="style"` would be assigned an internal request priority of *Highest*. The `as` attribute is required for caching the resource for future requests and applying the correct Content Security Policy<sup>908</sup>.

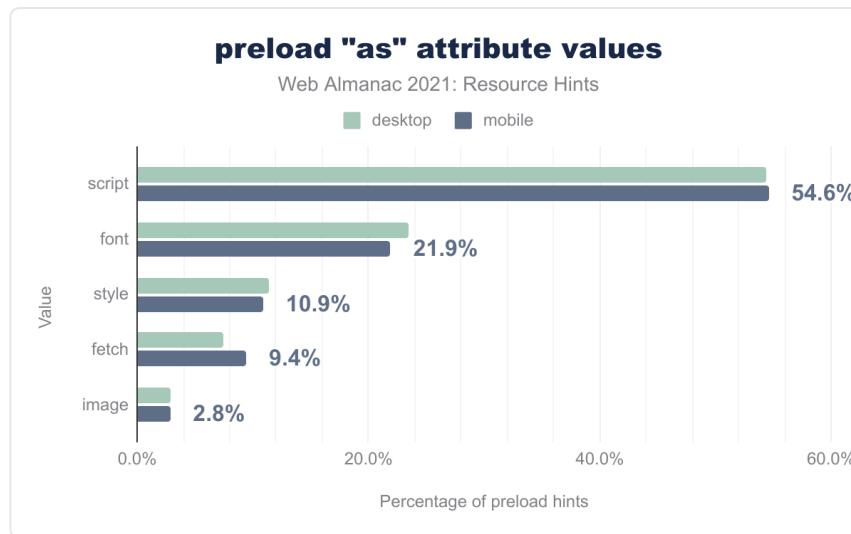


Figure 20.9. `rel="preload" as` attribute values.

### `script`

`script` is the most common value by a significant margin. `<script>` elements are usually discovered early as they are embedded in the initial HTML document, but it is a common practice to place `<script>` elements before the closing `</body>` tag. Since HTML is parsed sequentially, this means that the scripts will be discovered after the DOM is downloaded and parsed—and with more websites dependent on JavaScript frameworks, the necessity to have JavaScript load early has increased. The downside is that JavaScript resources would be prioritized over the other resources discovered within the HTML document, including images and stylesheets, possibly compromising the user experience.

908. <https://developer.mozilla.org/docs/Web/HTTP/CSP>

**font**

The second most commonly preloaded resource is the `font`, which is a late-discovered resource since the browser will only download a font file after the layout phase when the browser knows that the font will be rendered on the page.

**style**

Stylesheets are ordinarily embedded in the document's `<head>` and discovered early during the document parsing. Additionally, as stylesheets are render-blocking resources they are assigned the *Highest* request priority. This should make preloading stylesheets unnecessary, but it is sometimes required to re-prioritize the requests. A bug<sup>909</sup> in Google Chrome (fixed in Chrome 95) prioritizes preloaded resources ahead of other higher-priority resources discovered by the preload scanner, including CSS files. Preloading the stylesheet will restore its *Highest* priority. Another instance when stylesheets are preloaded is when they are not downloaded directly from the HTML document, such as the asynchronous CSS<sup>910</sup> “hack” which uses an `onload` event to avoid render-blocking the page with non-critical CSS.

**fetch**

Preload may be used to initiate a request to retrieve data which you know is critical to the rendering of the page, such as a JSON response or stream.

**image**

Preloading images may help improve the LCP score when the image is not included in the initial HTML, such as a CSS `background-image`.

**The `crossorigin` attribute**

The `crossorigin` attribute is used to indicate whether Cross-Origin Resource Sharing<sup>911</sup> (CORS) must be used when fetching the requested resource. This could apply to any resource type, but it is most commonly associated with font files as they should always be requested using CORS.

909. <https://bugs.chromium.org/p/chromium/issues/detail?id=629420>

910. <https://www.filamentgroup.com/lab/async-css.html>

911. <https://developer.mozilla.org/docs/Web/HTTP/CORS>

<b>value</b>	<b>desktop</b>	<b>mobile</b>
not set	66.6%	65.9%
<i>crossorigin</i> (or equivalent)	14.5%	13.5%
<i>use-credentials</i>	< 0.1%	< 0.1%

Figure 20.10. `rel="preload"` `crossorigin` attribute values.

### anonymous

The default value when no value is specified is `anonymous` and this value will set the credentials flag to `same-origin`. It is required when downloading resources protected by CORS. It is also a requirement<sup>912</sup> when downloading font files—even if they are on the same origin! If you omit the `crossorigin` attribute when the eventual request for the preloaded resource uses CORS, you will end up with a duplicate request since it won't match in the preload cache.

### use-credentials

When requesting cross-origin resources which require authentication, for example through the use of cookies, client certificates or the `Authorization` header; setting the `crossorigin="use-credentials"` attribute will include this data in the request and allow the server to respond to the request so that the resource may be preloaded. This is not a common scenario with 0.1% usage, however if your page content is dependent on an authenticated status, it could be used to initiate an early fetch request to get the login status.

## The `media` attribute

An oft-neglected feature available to `rel="preload"` is the ability to specify media queries through the `media` attribute—with less than 4% of all preloads using this attribute. The `media` attribute accepts media queries allowing you to target the media type and specific browser features, such as viewport width. As an example, the `media` attribute would allow you to preload a low-resolution image on devices with a narrow viewport and a full-sized image on devices with a large viewport.

In addition to the `media` attribute, the `<link>` element supports `imagesrcset` and

<sup>912</sup> <https://drafts.csswg.org/css-fonts/#font-fetching-requirements>

`imagesizes` attributes which correspond to the `srcset` and `sizes` attributes on `<img>` elements. Using these attributes, you can use the same resource selection criteria that you would use on your image. Unfortunately, their adoption is very low (less than 1%); most likely owing to the lack of support<sup>913</sup> on Safari.

*Note: The `media` attribute is not available on all `<link>` elements as the spec suggests, but it is only available on `rel="preload"`.*

## Bad practices

Owing to the versatility of `rel="preload"`, there isn't a clear set of rules dictating how to implement the preload hint, but we can learn a lot from our mistakes and understand how to avoid them.

### Unused preloads

We have already seen that there is a negative correlation between a website's performance and the number of preload hints. This relationship may be influenced by two factors:

- Incorrect preloads
- Unused preloads

An incorrect preload refers to when you preload a resource which is not as important as the other resources which the browser would have otherwise prioritized. We are unable to measure the extent of incorrect preloads as you would need to A/B test the page with and without each hint.

An unused preload occurs when you preload a resource which is not needed within the first few seconds of loading the page.

# 21.5%

Figure 20.11. Percent of unused preload hints within the first 3 seconds.

In such cases, the preload hint is regressing the website's performance, as you are instructing the browser to download and prioritize files or resources which are not needed immediately—or even not needed at all. This is one of the challenges when using resource hints,

<sup>913</sup>. [https://caniuse.com/mdn-html\\_elements\\_link\\_imagesizes](https://caniuse.com/mdn-html_elements_link_imagesizes)

as they require regular maintenance and automating the process opens the door to allow such issues to creep in.

### Incorrect `crossorigin` attribute

Attempting to preload a CORS-enabled resource without including the correct `crossorigin` attribute will download the same resource twice. The `crossorigin` attribute is required on the `<link>` element if the eventual request would also use CORS. This is also the case when requesting font files, even when self-hosting font files on the same origin, as font files are always treated as CORS-enabled.

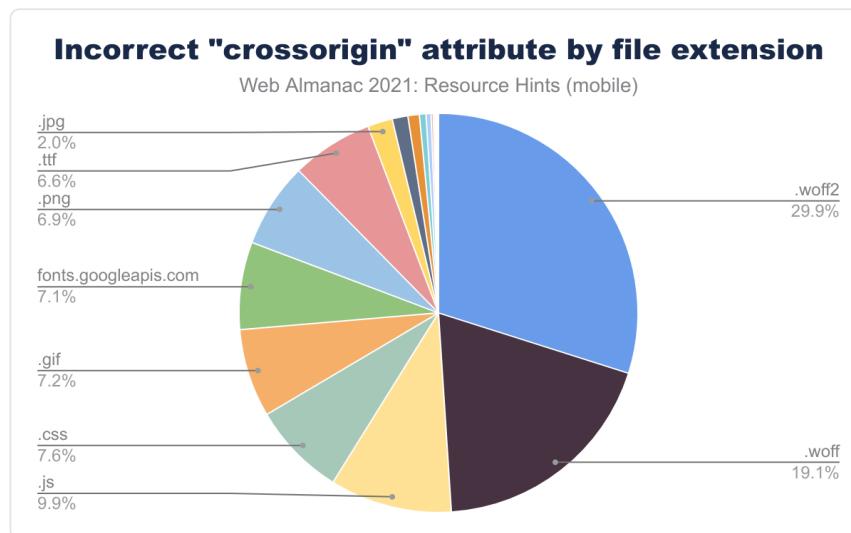


Figure 20.12. Percent of incorrect `crossorigin` values segmented by file extension on mobile devices.

More than half (63.6%) of the cases when the `crossorigin` attribute on the `rel="preload"` hint is either missing or incorrect, are linked to the preloading of font files, with a total of 14,818 instances across the dataset.

### Invalid `as` attribute

The `as` attribute plays an important role when preloading your resources and getting this wrong may result in downloading the same resource twice. On most browsers, specifying an unrecognized `as` attribute will ignore the preload. The supported values are `audio`, `document`, `embed`, `fetch`, `font`, `image`, `object`, `script`, `style`, `track`, `worker`

and `video`.

There are 17,861 cases of unrecognized values, with the most frequent error being omitting it completely; while the most common invalid `as` values are `other` and `stylesheet` (the correct value is `style`).



Figure 20.13. Pages incorrectly used `as="stylesheet"` instead of `"style"`

When using an incorrect `as` attribute value—as opposed to unrecognized value, such as using `style` instead of `script`—the browser will duplicate the file download as the request won't match the resource stored in the preload cache.

*Note: While `video` is included in the spec, it isn't supported by any browser and would be treated as an invalid value and ignored.*

### Unused font files

More than 5% of pages which preload font files preload more font files than needed. When preloading font files, all browsers which support `preload` also support `.woff2`. This means that, assuming that the `.woff2` font files are available, it is not necessary to preload older formats, including `.woff`.

## Third parties

You can use resource hints to connect to, or download resources from, both first and third parties. While `dns-prefetch` and `preconnect` are only useful when connecting to different origins, including subdomains, `preload` and `prefetch` may be used for both resources on the same origin and resources hosted by third parties.

When considering which resource hints you should use for third-party resources, you need to evaluate the priority and role of each third party on your application's loading experience and whether the costs are justified.

Prioritizing third-party resources over your own content is potentially a warning sign, however there are cases when this is recommended. As an example, if we look at cookie notice scripts—which are required in the European Union by General Data Protection

Regulation<sup>914</sup>—these are usually accompanied by a `dns-prefetch` or `preconnect` hint as they are highly obtrusive to the user experience and also a prerequisite for some site functions, such as serving personalized ads.

<b>host</b>	<b><code>dns-prefetch</code></b>	<b><code>preconnect</code></b>	<b><code>preload</code></b>	<b>Total</b>
<code>adservice.google.com</code>	0.2%	0.5%	35.7%	36.4%
<code>fonts.gstatic.com</code>	0.9%	24.0%	0.6%	25.5%
<code>fonts.googleapis.com</code>	14.0%	4.5%	2.7%	21.2%
<code>s.w.org</code>	19.7%	0.2%	-	19.9%
<code>cdn.shopify.com</code>	-	1.7%	9.6%	11.2%
<code>siteassets.parastorage.com</code>	-	-	5.9%	5.9%
<code>www.google-analytics.com</code>	1.2%	3.9%	0.2%	5.3%
<code>www.googletagmanager.com</code>	1.9%	2.7%	0.2%	4.8%
<code>static.parastorage.com</code>	-	-	4.7%	4.7%
<code>ajax.googleapis.com</code>	2.2%	1.6%	0.3%	4.1%
<code>www.google.com</code>	2.7%	1.0%	0.1%	3.8%
<code>images.squarespace-cdn.com</code>	-	3.5%	-	3.5%
<code>cdnjs.cloudflare.com</code>	1.6%	1.0%	0.4%	2.9%
<code>monorail-edge.shopifysvc.com</code>	2.0%	0.8%	-	2.8%
<code>fonts.shopifycdn.com</code>	-	1.1%	1.0%	2.1%

Figure 20.14. Most popular third-party connections using resource hints on mobile devices.

Analyzing the table above, 36.7% of all pages which include a `preload` hint are preloading resources hosted on `adservice.google.com`. The `s.w.org` host is the most popular domain for `dns-prefetch` and is used on WordPress sites (since version 4.6) for the loading of SVG images from its Twemoji CDN, when the browser is detected to not support native emoji characters. Google Fonts related services on `fonts.gstatic.com` and `fonts.googleapis.com` are the two most popular hosts for the `preconnect` directive.

914. [https://en.wikipedia.org/wiki/General\\_Data\\_Protection\\_Regulation](https://en.wikipedia.org/wiki/General_Data_Protection_Regulation)

### Use on the web

To embed a font, copy the code into the `<head>` of your html

`<link>`  `@import`

```
<link rel="preconnect" href="https://fonts.googleapis.com">
<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
<link href="https://fonts.googleapis.com/css2?family=Roboto&display=swap" rel="stylesheet">
```

Figure 20.15. Google Fonts instructions to preconnect to fonts.gstatic.com and fonts.googleapis.com. (Source: Google Fonts<sup>915</sup>)

Google Fonts now includes instructions to `preconnect` to both the fonts.gstatic.com origin and fonts.googleapis.com, which is usually good practice to offset the impact of these late discovered resources.

To learn more about the state of third parties, check out the Third Parties chapter.

## Native lazy-loading

Lazy-loading refers to the technique to defer downloading a resource—in this case an image or iframe—until it is needed or visible within the viewport. Native lazy-loading refers to the ability to specify this in the HTML with a `loading="lazy"` attribute, rather than having to use a JavaScript library to handle this. Native image and iframe lazy-loading have been standardized in 2019 and since then their adoption—especially for images—has grown exponentially.

`loading="lazy"` for images is supported on most major browsers. On Safari, it is marked as in progress<sup>916</sup> and is available behind a flag, but not yet enabled by default.

Lazy-loading of iframes is supported on Chrome, once again behind a flag on Safari but not yet supported on Firefox<sup>917</sup>.

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

916. [https://bugs.webkit.org/show\\_bug.cgi?id=200764](https://bugs.webkit.org/show_bug.cgi?id=200764)

917. [https://bugzilla.mozilla.org/show\\_bug.cgi?id=1622090](https://bugzilla.mozilla.org/show_bug.cgi?id=1622090)

Browsers which do not support the `loading` attribute will simply ignore it—making it safe to add without unwanted side-effects. JavaScript based alternatives, such as `lazysizes`<sup>918</sup> may still be used, however considering that full browser support is around the corner, it may not be worth adding to a project at this stage.



Figure 20.16. The percent of pages that have the `loading="lazy"` attribute on `img` elements.

The percent of pages using `loading="lazy"` has grown from 4.2% in 2020 to 17.8% by the time of our analysis. That's a whopping 423% growth! This rapid growth is extraordinary and is likely driven by two key elements: the ease with which it could be added to pages without cross-browser compatibility issues, and the frameworks or technologies powering these websites. In WordPress 5.5, lazy-loading images became the default implementation<sup>919</sup>, supercharging the adoption rate of `loading="lazy"`, with WordPress sites now making up 84%<sup>920</sup> of all pages which use native image lazy-loading.

918. <https://github.com/aFarkas/lazysizes>

919. <https://make.wordpress.org/core/2020/07/14/lazy-loading-images-in-5-5/>

920. <https://web.dev/lcp-lazy-loading/>

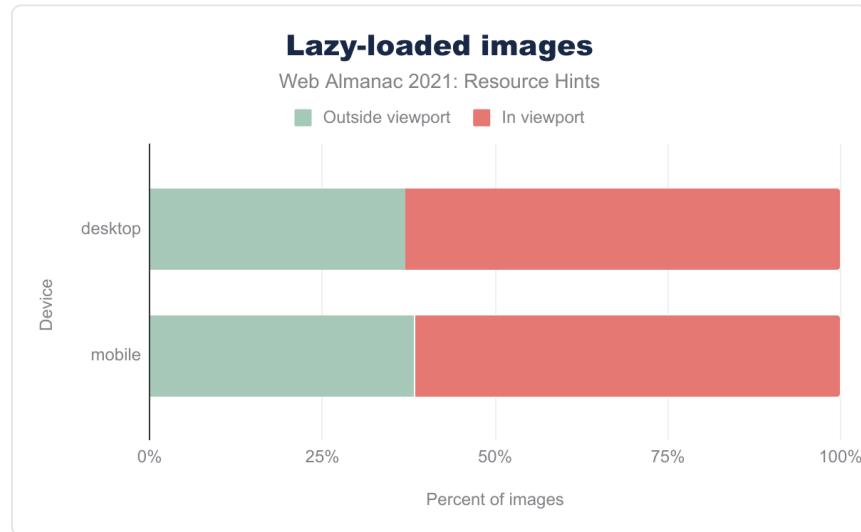


Figure 20.17. Percent of `img` elements with `loading="lazy"` which are in the initial viewport.

61.5% of lazy-loaded images on mobile and 63.1% of lazy-loaded images on desktop are actually within the initial viewport and shouldn't be lazy-loaded. A study<sup>921</sup> on the load times for pages which use lazy-loading indicated that pages which use lazy-loading tend to have a worse LCP performance, possibly caused by overusing the `lazy`-loading attribute. This is increasingly significant on the LCP element, which shouldn't be lazy-loaded. If you are using `loading="lazy"`, you should check that the lazily-loaded images are below the fold and more critically, that the LCP element is not lazy-loaded. You may dig deeper into the effects of lazy-loading the LCP image on your Core Web Vitals in the Performance chapter.

# 2.6%

Figure 20.18. Percent of pages that have the `loading="lazy"` attribute on `iframe` elements.

The likelihood of a page containing at least one `iframe` is much lower than for that containing an image with only 2.6% of pages containing an `iframe` taking advantage of native lazy-loading. The benefits of lazy-loading an `iframe` are potentially important, as an `iframe` could initiate further requests to download even more resources, including scripts and images. This is especially true when using embeds, such as YouTube or Twitter embeds. Similarly, when deciding the loading strategy for an image, you must check whether the `iframe` is shown within the initial viewport

<sup>921.</sup> <https://web.dev/lcp-lazy-loading/>

or not. If it isn't, then it is usually safe to add `loading="lazy"` to the `<iframe>` element to benefit from a reduced initial load and boost performance.

## HTTP/2 Server Push

HTTP/2 supports a technology called *Server Push* that preemptively pushes a resource it expects the client will be requesting. As the server is pushing the resource instead of informing the client that it should request it, cache-management becomes complex and, in some cases, the pushed resources would even delay the delivery of the HTML, which is critical for discovering all resources required to load the page.

Unfortunately, HTTP/2 push has been disappointing, with little evidence that it provides the performance boost promised compared to the risk of over pushing resources that either the browser already has, or that are of less importance than resources the browser requests.

So, while the technology is widely available, overcoming these obstacles makes it highly unpopular—with less than 1% adoption. Chrome has also filed an Intent to Remove<sup>922</sup> that is paused until a testable implementation of 103 Early Hints (covered next) is available. Chrome does not support<sup>923</sup> Server Push on HTTP/3 either.

You can read more about HTTP, HTTP/2, and HTTP/3 in the HTTP chapter.

## Future

While there are no proposals to add new `rel` directives, improvements from the browser vendors to the current set of resource hints—such as the prioritization bug<sup>924</sup> in Chrome—are expected to have a positive impact. Hint adoption is expected to evolve, and the use of `preload` should shift towards its intended purpose: late discovered resources.

Additionally, two proposals, 103 Early Hints and Priority Hints, are expected to be made available soon, with experimental support already available on Chrome.

## 103 Early Hints

Chrome 95 added experimental support for 103 Early Hints<sup>925</sup> for `preload` and `preconnect`. Early hints enable the browser to preload resources before the main response is served and

922. <https://lists.w3.org/Archives/Public/ietf-http-wg/2019JulSep/0078.html>

923. <https://github.com/httplib/httplib2-spec/issues/78#issuecomment-724371629>

924. <https://bugs.chromium.org/p/chromium/issues/detail?id=629420>

925. <https://datatracker.ietf.org/doc/html/rfc8297>

take advantage of the idle time on the browser between the request being sent and the response from the server. When using 103 Early Hints, the server immediately sends an “informational” response status detailing the resources to be preloaded using the HTTP header method, while processing the real document response. This way, the browser will be able to initiate preload requests for critical resources even before the HTML arrives and much earlier than it would if using the `<Link>` element in the document markup. 103 Early Hints overcomes most of the difficulties encountered with HTTP/2 Server Push.

## Priority Hints

Priority hints inform the browser of the relative importance of resources within the page, intending to prioritize critical resources and improve Core Web Vitals. Priority Hints are enabled through the document markup by adding the `importance` attribute to resources, such as `<img>` or `<script>`. The `importance` attribute accepts an enumeration of `high`, `low` or `auto` and by combining this with the type of resource, the browser would be able to assign the optimal fetch priority based on its heuristics. Priority Hints are available on Chrome 96 as an origin trial<sup>926</sup>.

## Conclusion

During the past year, resource hint adoption grew and is expected to continue growing as developers take advantage of these APIs to prioritize resources and improve the user’s experience. At the same time, browser vendors have continued calibrating these directives, evolving their role and effectiveness.

Resource hints could become a double-edged sword if the benefit for your users is not evaluated. Almost a quarter of preload requests went unused while the number of preload hints correlated with slower load times.

Resource hints are akin to fine-tuning a race car’s engine. They would not turn a slow engine into a fast one, and too many adjustments could break it. Yet, some small tweaks here and there would allow you to maximize it.

So once again, the mantra behind resource hints remains, “if everything is important, then nothing is”. Use resource hints wisely and don’t overuse them.

926. <https://developer.chrome.com/blog/origin-trials/>

## Author

---



### Kevin Farrugia

Twitter @imkevdev GitHub kevinfarrugia Website <https://imkev.dev>

Kevin Farrugia is a consultant on web performance and software architecture. You can find him blogging on [imkev.dev](https://imkev.dev)<sup>927</sup>.

---

---

<sup>927.</sup> <https://imkev.dev>

# Part IV Chapter 21

## CDN



*Written by Navaneeth Krishna*

*Reviewed by Julia Yang and Shilpa Raghunathan*

*Analyzed by Paul Calvano*

*Edited by Julia Yang and Shaina Hantsis*

### Introduction

A Content Delivery Network (CDN) is a geographically distributed network of proxy servers in data-centers. The goal of a CDN is to provide high availability and performance for web content. It does this by distributing content closer to the end users.

CDNs have been in existence for over two decades. With the exponential rise in internet traffic contributed by online video consumption, online shopping, and increased video conferencing due to COVID-19, CDNs are required more than ever before. They ensure high availability and good web performance despite this growth in internet traffic.

During the early days, a CDN was a simple network of proxy servers which would:

1. Cache content (like HTML, images, stylesheets, JavaScript, videos, etc.)
2. Reduce network hops for end users to access content
3. Offload TCP connection termination away from the data centers hosting the web

### properties

They primarily helped web owners to improve the page load times and to offload traffic from the infrastructure hosting these web properties.

Over time, the services offered by CDN providers have evolved beyond caching and offloading bandwidth/connections. Now they offer additional services such as:

- Cloud-hosted Web Application Firewalls
- Bot Management solutions
- Clean pipe solutions (Scrubbing Data-centers)
- Serverless Computing offerings
- Image and Video Management solutions etc.,

Thus, a web owner these days has a lot of options to choose from. This can be overwhelming and complex since these new offerings from CDNs make them an extension of your application and require closer integration with application development life-cycles.

There are benefits to web owners in pushing web application logic and workflows closer to the end user. This eliminates the round trip and bandwidth that a HTTP/HTTPS request would take. It also handles near-instant scalability requirements for the origin. A side-effect of this is that Internet Service Providers (ISPs) benefit from the scalability management as well, which improves their infrastructure capacities.

This reduction in requests reduces the load on the internet backbone, (read Middle-Mile of the Internet<sup>928</sup>). It also helps manage more of the internet load within the last mile of the internet. Thus, a CDN plays a multifaceted role in the Internet landscape as it allows web owners to improve the performance, reliability and scalability of content delivery.

## Caveats and disclaimers

As with any observational study, there are limits to the scope and impact that can be measured. The statistics gathered on CDN usage for the Web Almanac are focused more on applicable technologies in use and not intended to measure performance or effectiveness of a specific CDN vendor. While this ensures that we are not biased towards any CDN vendor, it also means that these are more generalized results.

These are the limits to our testing methodology:

928. [https://en.wikipedia.org/wiki/Middle\\_mile](https://en.wikipedia.org/wiki/Middle_mile)

- **Simulated network latency:** We use a dedicated network connection that synthetically shapes traffic.
- **Single geographic location:** Tests are run from a single datacenter and cannot test the geographic distribution of many CDN vendors.
- **Cache effectiveness:** Each CDN uses proprietary technology and many, for security reasons, do not expose cache performance.
- **Localization and internationalization:** Just like geographic distribution, the effects of language and geo-specific domains are also opaque to these tests.
- **CDN detection:** This is primarily done through DNS resolution and HTTP headers. Most CDNs use a DNS CNAME to map a user to an optimal datacenter. However, some CDNs use Anycast IPs or direct A+AAAA responses from a delegated domain which hide the DNS chain. In other cases, websites use multiple CDNs to balance between vendors, which is hidden from the single-request pass of our crawler.

All of this influences our measurements.

Most importantly, these results reflect the utilization of specific features (Example: TLS, HTTP/2 etc.) per site, but do not reflect actual traffic usage. YouTube is more popular than “www.example.com” yet both will appear as equal value when comparing utilization.

With this in mind, here are a few statistics that were intentionally not measured in the context of a CDN:

1. Time To First Byte (TTFB)
2. CDN Round Trip Time
3. Core Web Vitals
4. Cache Hit versus Cache Miss performance etc.

While some of these could be measured with HTTP Archive dataset, and others by using the CrUX dataset, the limitations of our methodology and the use of multiple CDNs by some sites, will be difficult to measure and so could be incorrectly attributed. For these reasons, we have decided not to measure these statistics in this chapter.

## CDN adoption

The contents in a web page can be divided into 3 parts, namely:

1. Base HTML page (e.g., `www.example.com`)
2. Embedded first-party content on subdomains (e.g., `images.example.com`, `css.example.com` etc.)
3. Third-party content (e.g., Google Analytics, Advertisements etc.)

From their inception, CDNs have been the go-to solution for delivering embedded content such as images, stylesheets, JavaScript, and fonts. This kind of content doesn't change frequently, making it a good candidate for caching on a CDN's proxy servers.

With the evolution of CDN technology an expressway was set up on the internet for non-cacheable assets. This means the main web page and APIs can now be delivered reliably and faster, compared to a TCP connection to the origin.

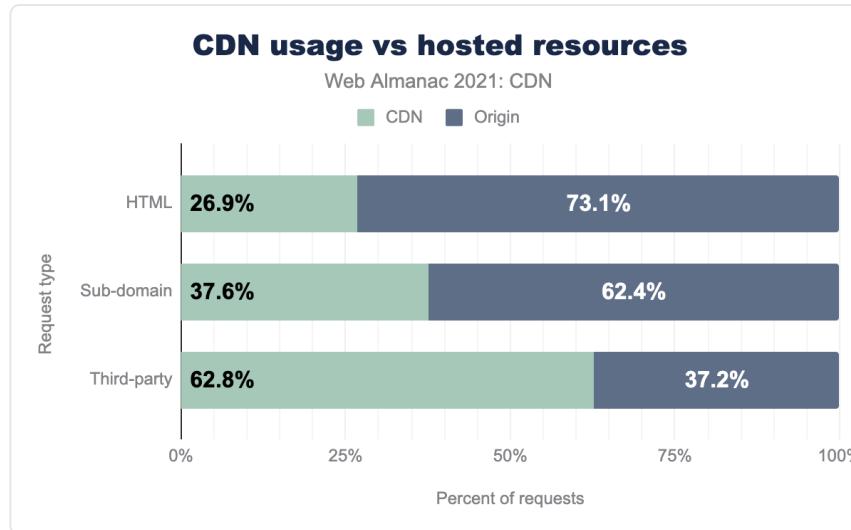


Figure 21.1. CDN usage vs hosted resources.

The impact of this can be seen in the above chart when we compare this against the same data in 2019 chapter<sup>229</sup> (note, there was no CDN chapter in 2020 Web Almanac). It's good to see the trend of sites using CDN has improved by 7% between 2019 and 2021. This shows that more of the industry is leveraging CDNs to take benefit of consistent content delivery times and minimize the impact of congestion on Internet.

229. <https://almanac.httparchive.org/en/2019/cdn>

Looking at third-party content, there is negative growth for CDN adoption. Compared to 2019 chapter<sup>930</sup>, we see 3% reduction in domains using CDNs. Third-party domains are used by SaaS vendors for analytics, advertisements, responsive pages, etc. It is in the SaaS vendor's interest to use CDNs for their services. Their content is used by multiple web owners and this content gets accessed by end users across geographies, making CDNs necessary from both a business and performance standpoint. This is evident in the charts where it's clear that third-party content has the highest adoption of CDN.

But why do we see this negative growth in CDN Adoption for third-party domains?

The probable reasons for this include:

- The HTTP/2 protocol requires web owners to consolidate the domains instead of using multiple domains for optimal performance
- Contribution of third-party content to total page weight has also increased over the years (refer to the Third Parties chapter for more details) leading to increased page load time concerns for web owners
- Customization/personalization of third-party scripts to suit the requirements of web owners

These changes have led to the SaaS vendors offering "self-hosting" options to web owners. This leads to more content being delivered over the first-party domain instead of the vendor's domain. When this happens, it's up to the web owner to either deliver the content over a CDN or directly from their hosting infrastructure.

While we observed CDN adoption across different types of content, we will look at this data from a different point of view below.

930. <https://almanac.httparchive.org/en/2019/cdn>

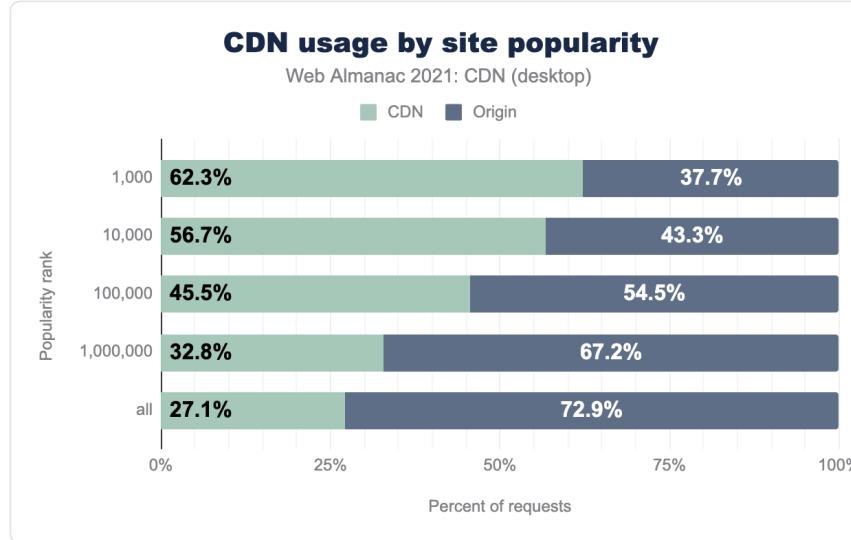


Figure 21.2. CDN usage by site popularity (desktop).

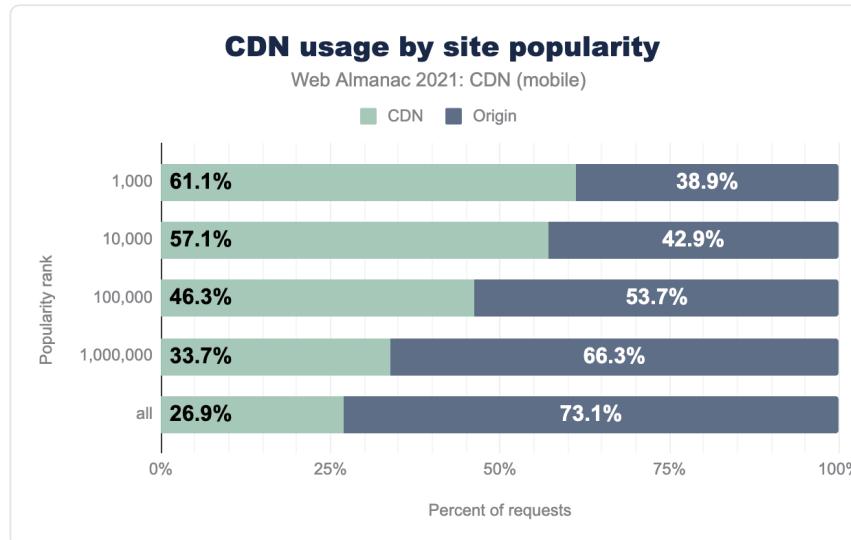


Figure 21.3. CDN usage by site popularity (mobile).

Ranking the websites based on their popularity (sourced from Google's Chrome UX Report) in the web and then checking for their CDN usage, the top 1,000 contribute to the highest usage of CDN. The top websites are owned by larger companies like Google and Amazon, who contribute to much of the internet traffic we see today, so it's no surprise that these names

make it to the list of top CDN providers in the next section. This also backs the fact about the benefits CDNs bring to the table when operating at scale and having the ability to scale further if needed.

# 61.1%

*Figure 21.4. Percent of top 1,000 mobile websites using a CDN.*

The CDN adoption rate falls below 50% when we look at the top 100,000 websites but the rate of reduction slows down beyond this. For the full data set (which is 6.2 million sites on desktop and 7.5 million on mobile), 27% of these websites use CDN. When you translate that percentage into real number, that's 2 million mobile websites using CDN! It's not such a small number when you look at it this way.

But the decreasing percentage of CDN adoption in the low-popularity website end does make sense considering the benefits of CDN (such as caching and TCP connection offload) increases with the number of end users on the web property. Below a certain scale of end-user traffic on a web property, the cost-to-benefit math of a CDN may not work in web property owner's favor and they might be better off delivering the web content directly from the origin.

## Top CDN providers

CDN providers can be broadly classified into 2 segments:

1. Generic CDN (Akamai, Cloudflare, Fastly etc.)
2. Purpose-built CDN (Netlify, WordPress etc.)

Generic CDN addresses the mass market requirements. Their offerings include:

- Web site delivery
- mobile app API delivery
- Video streaming
- Serverless compute offerings
- Web security offerings, etc.

This appeals to a larger set of industries and is reflected in the data. Generic CDNs hold the

lion's share of the HTML and First party subdomain traffic:



Figure 21.5. Top CDNs for HTML requests.

CDN providers such as Cloudflare, Fastly, Akamai and Limelight appear in this list of Generic CDN providers. We also see other providers such as Google and AWS. They appear in this list since they offer bundled CDN offerings along with their Cloud hosting services. These bundles help reduce load on the hosting infrastructure and also improves web performance.



Figure 21.6. Top CDNs for sub-domain requests.

Looking at third-party domains below, a different trend in top CDN providers is seen. We see Google top the list before the generic CDN providers. The list also brings Facebook into prominence. This is backed by the fact that a lot of third-party domain owners require CDNs more than other industries. This necessitates them to invest in building a purpose-built CDN. A purpose-built CDN is one which is optimized for a particular content delivery workflow.



Figure 21.7. Top CDNs for third-party requests.

For example, a CDN built specifically to deliver advertisements will be optimized for:

- High input-output (I/O) operations
- Effective management of long tail<sup>931</sup> content
- Geographical closeness to businesses requiring their services

This means purpose-built CDNs meet the exact requirements of a particular market segment as opposed to a generic CDN solution. Generic solutions can meet a broader set of requirements but are not optimized for any particular industry or market.

## TLS adoption impact

With CDNs set up in the request-response workflows, the end-user's TLS connection terminates at the CDN. In turn, the CDN sets up a second independent TLS connection and this connection goes from the CDN to the origin host. This break in the CDN workflow allows the CDN to define the end-user's TLS parameters. CDNs tend to also provide automatic updates to internet protocols. This allows web owners to receive these benefits without making changes to their origin.

<sup>931.</sup> [https://en.wikipedia.org/wiki/Long\\_tail](https://en.wikipedia.org/wiki/Long_tail)

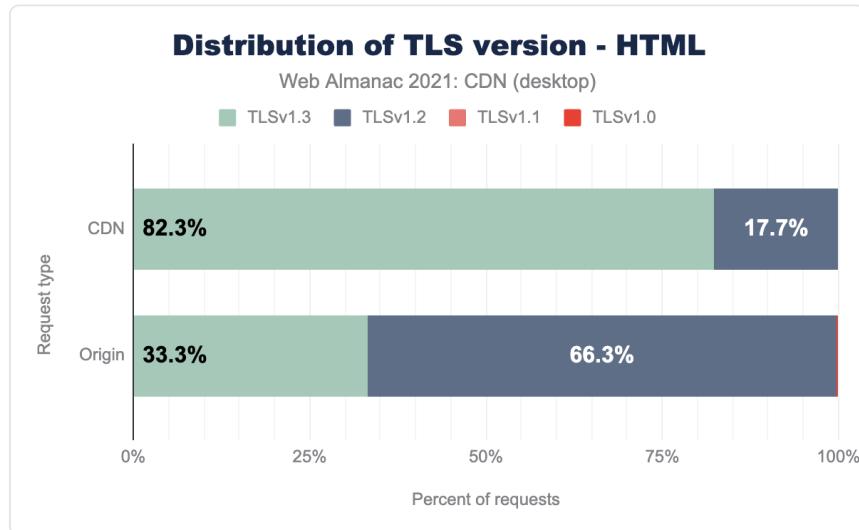


Figure 21.8. Distribution of TLS version for HTML (desktop).

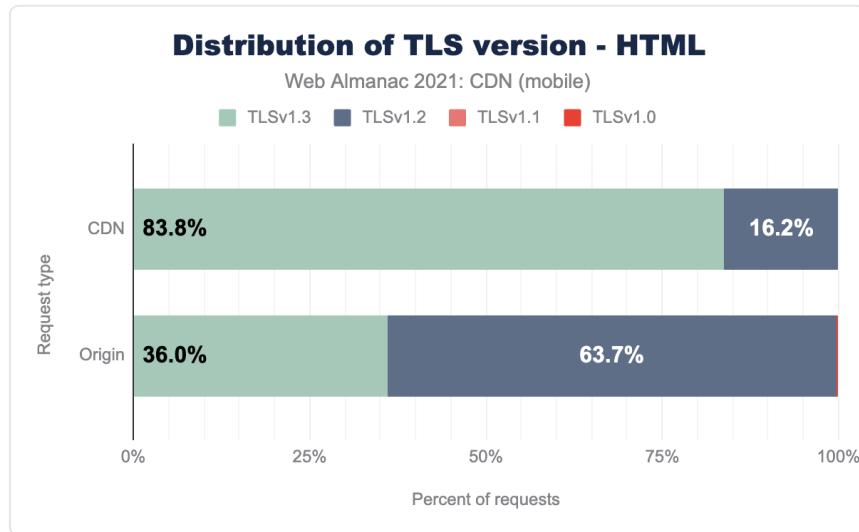


Figure 21.9. Distribution of TLS version for HTML (mobile).

We see in the data above that 83% websites on CDNs use TLS 1.3 compared to 33-36% on the origin. That's a huge benefit of using a CDN. These protocol upgrades also come with minimal to no-effort for web owners. The trend is identical for mobile and desktop websites.

Similar trend is observed for the third-party domains below. These web services with CDNs

have better adoption of TLS 1.3 than the ones without for the same reasons.

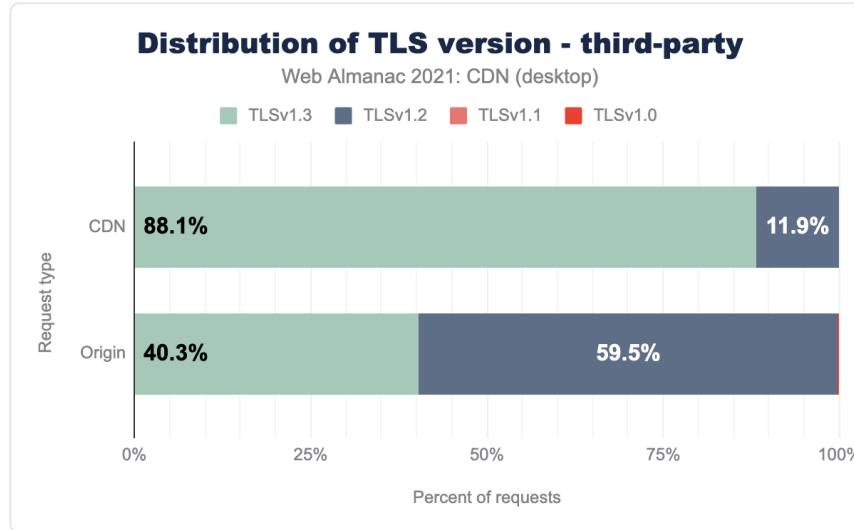


Figure 21.10. Distribution of TLS version for third-party requests (desktop).

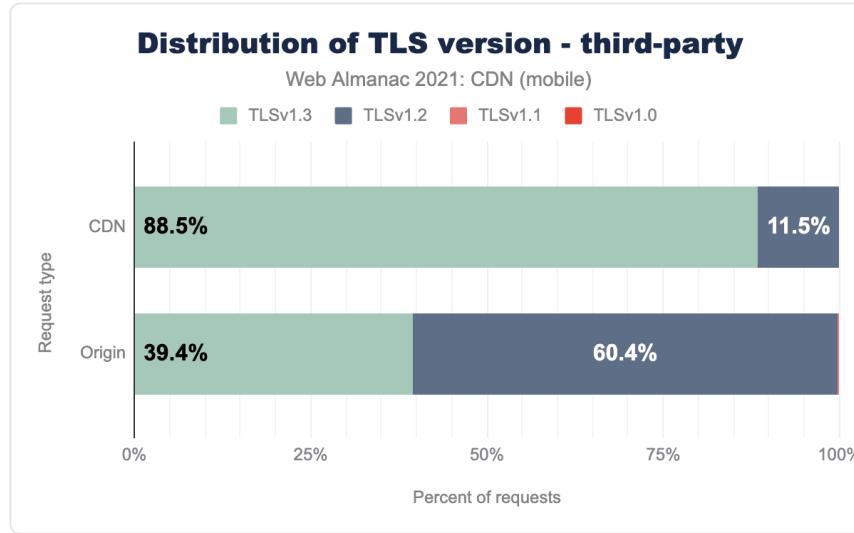


Figure 21.11. Distribution of TLS version for third-party requests (mobile).

It is important for third-party domains to be on the latest TLS version for security reasons. With the increase in web attacks, web owners are aware of loopholes that can be exploited with unsecure connections to third-party domains. They will expect equally secure TLS connections

which meet the security and performance requirements of their web sites. These expectations enhance the benefits CDNs bring to the table.

## TLS performance impact

Common logic dictates that the fewer hops it takes for a HTTPS request-response to traverse, the faster the round trip would be. So exactly how much quicker can it be if the TLS connection terminates closer to the end user? The answer: As much as 3 times faster!

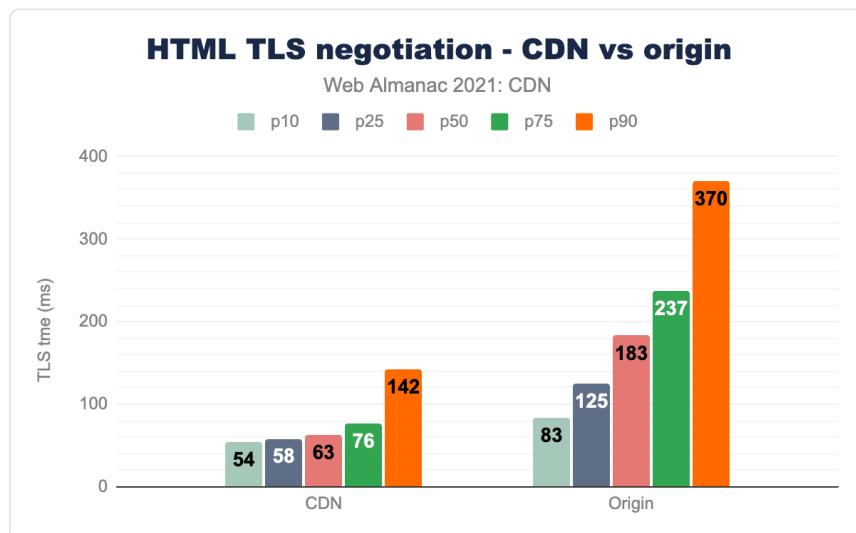


Figure 21.12. HTML TLS negotiation - CDN vs origin.

CDNs have helped slash the TLS connection times. This is due to their proximity to the end user and adoption of newer TLS protocols that optimize the TLS negotiation. CDNs hold the edge over origin at all percentiles here. At P10 and P25, CDNs are nearly 1.5x to 2x faster than origin in TLS set up time. The gap increases even more once we hit the median and above, where CDNs are nearly 3x faster. 90th percentile users using a CDN will have better performance than 50th percentile users on direct origin connections.

This is quite important when you consider that all sites will have to be on TLS these days. Optimal performance at this layer is essential for other steps that follow TLS connection. In this regard, CDNs are able to move more users to lower percentile brackets compared to direct origin connections.

## HTTP/2+ (HTTP/2 or better) adoption

HTTP/2 was introduced with a lot of hype and expectation. This was because the application layer protocol had not been updated since HTTP 1.1 in 1997. Since then, the web traffic trend, content-type, content size, website design, platforms, mobile apps and more have evolved significantly. Thus, there was a need to have a protocol which can meet the requirements of the modern-day web traffic and that protocol was realized with HTTP/2, and then further improved with the more recent HTTP/3.

However, the implementation challenges of HTTP/2 discouraged adoption. In addition, the net performance gains which can be expected with these changes was also not clear. Challenges repeated with the introduction of HTTP/3.

This was where the CDNs being the intermediary can help in bridging the challenge of HTTP/2 implementation for web owners. An HTTP/2 connection terminates at the CDN level, and this provides web owners the ability to deliver their website and subdomains over HTTP/2 without the need to upgrade their infrastructure to support it—the exact same reasons and benefits we saw for newer TLS versions.

CDNs act as the proxy to bridge the gap by providing a layer to consolidate hostnames and route traffic to relevant endpoints with minimal change to their hosting infrastructure. Features like prioritizing content in the queue and server push can be managed from the CDN's side and a few CDN's even provide hands-off automated solutions to run these features without any inputs from website owners, thus providing a boost to HTTP/2 adoption.

The trend cannot be clearer than what the graph shows below. There is high HTTP/2+ adoption by domains on CDNs compared to the ones not using a CDN.

*Note that due to the way HTTP/3 works (see the HTTP chapter for more information), HTTP/3 is often not used for first connections which is why we are instead measuring "HTTP/2+", since many of those HTTP/2 connections may actually be HTTP/3 for repeat visitors (we have assumed that no servers implement HTTP/3 without HTTP/2).*



Figure 21.13. Distribution of HTTP versions for HTML (desktop).



Figure 21.14. Distribution of HTTP versions for HTML (mobile).

Back in 2019, the origin domains had 27% adoption of HTTP/2 compared to 71% adoption on CDN. While we see in desktop sites that there is about a 14% increase in origins supporting HTTP/2+ in 2021, domains on CDNs have maintained that lead with a 15% increase. This gap is a bit less when we look at mobile sites, where domains using a CDN have a slightly lower HTTP/2+ adoption compared to desktop sites.

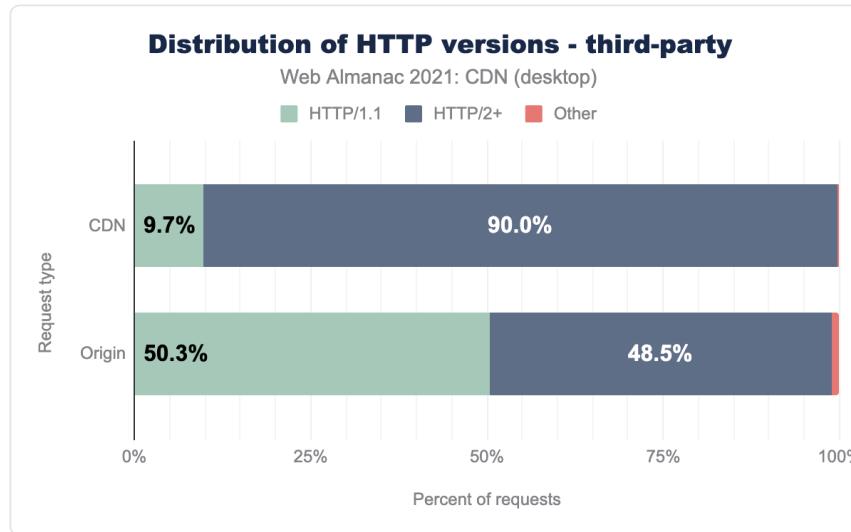


Figure 21.15. Distribution of HTTP versions for third-party requests (desktop).

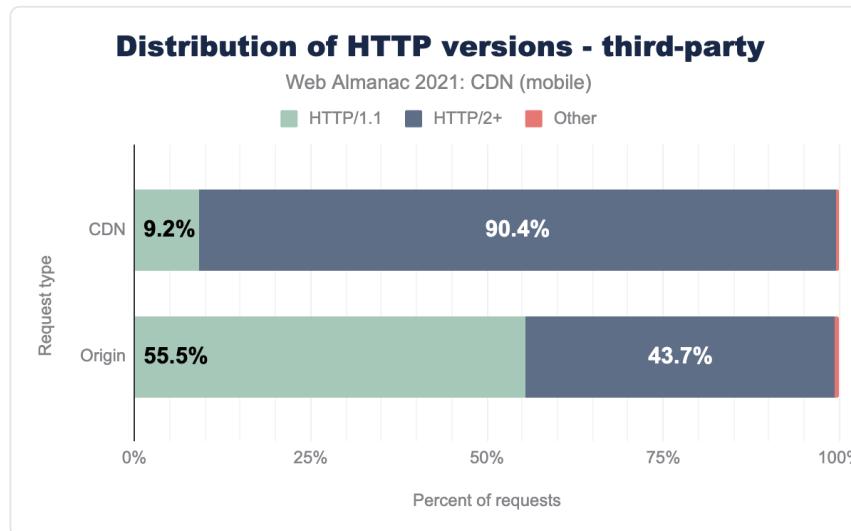


Figure 21.16. Distribution of HTTP versions for third-party requests (mobile).

Looking at third-party domains supporting newer protocols, we see an interesting trend of higher adoption of HTTP/2+protocols compared to first-party domains. This makes sense, considering the fact that most of the top third-party domains use purpose-built CDNs and thus have more control on the content development and content delivery. Additionally, third-party domains need to have consistent performance across all network conditions, and this is where

HTTP/2+ adds value by mixing in other protocols like UDP (used by HTTP/3) along with traditional TCP connections.

Back in 2019, Uber did an experiment to understand how UDP along with TCP (aka QUIC, the transport layer of HTTP/3) can help deliver content with consistent performance and overcome packet loss in highly congested mobile networks. The results of this experiment documented in this blog post<sup>932</sup> throws valuable insights into the demographic where HTTP/3 can help. Over time, this trend will trickle down and we should see web owners adopting HTTP/3, especially with mobile network traffic having a higher contribution to the total internet traffic.

## Brotli adoption

Content delivered over the internet employs compression to reduce the payload size. A smaller payload means it's faster to deliver the content from server to end user. This makes websites load faster and provide a better end-user experience. For images, this compression is handled by image file formats like JPEG, WEBP, AVIF, etc. (refer to the Media chapter for more on this). For textual web assets (such as HTML, JavaScript, and stylesheets) compression was traditionally handled by a file format called Gzip. Gzip has been in existence since 1992. It did a good job of making text asset payloads smaller, but a new text asset compression can do better than Gzip: *Brotli* (refer to the Compression chapter for more information).

Similar to TLS and HTTP/2 adoption, Brotli went through a phase of gradual adoption across web platforms. At the time of this writing, Brotli is supported by 96%<sup>933</sup> of the web browsers globally. However, not all websites compress text assets in Brotli format. This is because of both lack of support and of the longer time required to compress a text asset in Brotli format compared to Gzip compression. Also, the hosting infrastructure needs to have backward compatibility to serve Gzip compressed assets for older platforms which do not support the Brotli format, which can add complexity.

The impact of this is observed when we compare websites which are using CDN against the ones not using CDN.

932. <https://eng.uber.com/employing-quic-protocol/>

933. <https://caniuse.com/brotli>

## Distribution of compression types

Web Almanac 2021: CDN (desktop)

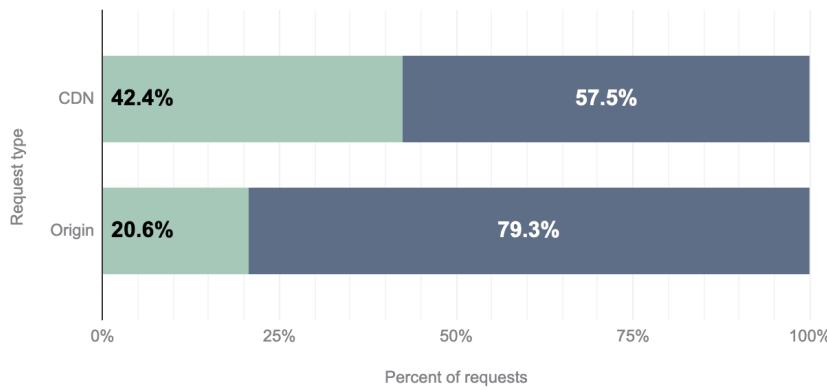



Figure 21.17. Distribution of compression types (desktop).

## Distribution of compression types

Web Almanac 2021: CDN (mobile)

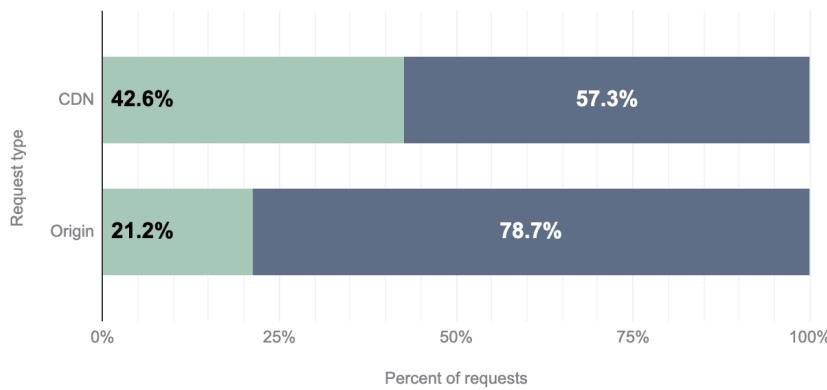



Figure 21.18. Distribution of compression types (mobile).

On both desktop and mobile platforms, we see that CDNs are delivering twice as many text assets in Brotli, compared to domains delivered from origin. From the CDN adoption section covered earlier, 73% of the domains serving sites are on CDNs and these can all benefit from the Brotli compression. By offloading the computational load of compressing a text asset in the

Brotli format to CDNs, website owners need not invest resources for hosting infrastructure.

However, it is at the web property owner's discretion whether to use Brotli compression on their CDNs or not. Compared to 95% of the web browsers globally which support Brotli compression, even with CDNs in place, less than half of all the text assets are delivered in Brotli format—so there is clearly space for this adoption to improve.

## Conclusion

There are limitations to the insights we can deduce about CDNs from the outside, since it is hard to know the secret sauce powering them behind the scenes. However, we have crawled the domains and compared the ones on CDNs against those who are not. We can see that CDNs have been an enabler for websites to adopt new web protocols, from the network layer to the application layer.

This impact is universal, with similar adoption rates across mobile and desktop: from using the latest TLS versions to upgrading to the newest HTTP versions (like HTTP/2, HTTP/3) to using the Brotli compression. What stands out is the depth of this impact and the sizable lead the CDN domains have built relative to non-CDN domains.

This role of CDNs is highly valuable and this will continue to be the case. CDN providers are also a key part of the Internet Engineering Task Force<sup>934</sup>, where they help shape the future of the internet. They will continue to play a key role aiding the internet-enabled industries to operate smoothly, reliably and quickly.

### Author



#### Navaneeth Krishna

@Navanee55755217 Navaneeth-akam

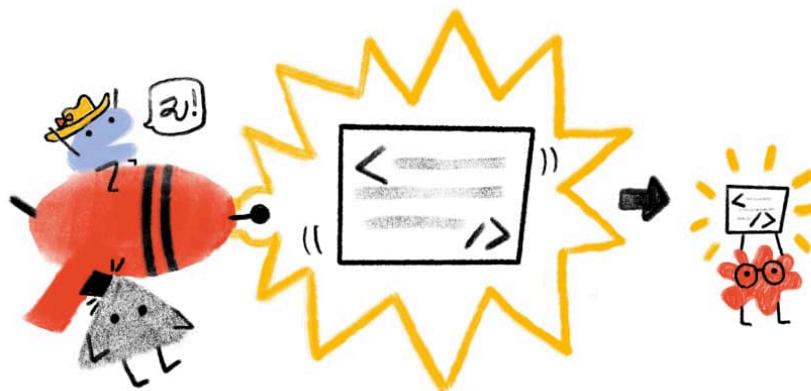
Navaneeth Krishna is a Web Performance Architect at Akamai<sup>935</sup>, a leading CDN provider. With over a decade of experience in the CDN industry, he believes the CDN will be an integral part to the growth of internet in the years to come and it will be a space to watch out for. You can find him tweeting @Navanee55755217.

934. <https://www.ietf.org/>  
 935. <https://www.akamai.com/>



# Part IV Chapter 22

# Compression



Written by Lode Vandevenne, Moritz Firsching, and Jyrki Alakuijala

Reviewed by Thomas Fischbacher, Eugene Kliuchnikov, and Iulia Comşa

Analyzed by Paul Calvano

Edited by Shaina Hantsis

## Introduction

A user's time is valuable, so they shouldn't have to wait a long time for a web page to load. The HTTP protocol allows the responses to be compressed, which decreases the time needed to transfer the content. Compression often leads to significant improvement in the user experience. It can reduce page weight, improve web performance and boost search rankings. As such, it's an important part of Search Engine Optimization.

This chapter discusses lossless compression applied on a HTTP response. Lossy and lossless compression used in media<sup>936</sup> formats such as images, audio and video are equally (if not more) important for increasing the page loading speed. However, these are not in the scope of this chapter, as they usually are part of the file format itself.

936. <https://almanac.httparchive.org/en/2020/media>

## Content types using HTTP compression

HTTP compression is recommended for text-based content, such as HTML, CSS, JavaScript, JSON, or SVG, as well as for `woff`, `ttf` and `ico` files. Media files such as images that are already compressed do not benefit from HTTP compression since, as mentioned previously, their representation already includes internal compression.

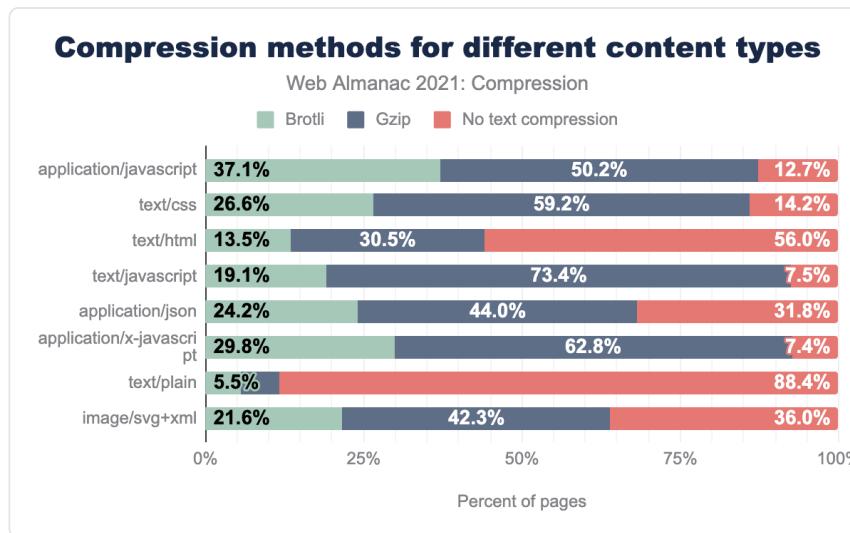


Figure 22.1. Compression methods for different content types

Compared to the other content types, `text/plain` and `text/html` use the least amount of compression, with merely 12% and 14% using compression at all. This might be because `text/html` is more often dynamically generated than static content such as JavaScript and CSS, even though compressing dynamically generated content also has a positive impact. More analysis about the compression of JavaScript content is available in the JavaScript chapter.

## Server settings for HTTP compression

For HTTP content encoding, the HTTP standard defines the `Accept-Encoding`<sup>937</sup> request header, with which a HTTP client can announce to the server what content encodings it can handle. The server's response can then contain a `Content-Encoding`<sup>938</sup> header field that specifies which of the encodings was chosen to transform the data in the response body.

937. <https://developer.mozilla.org/docs/Web/HTTP/Headers/Accept-Encoding>

938. <https://developer.mozilla.org/docs/Web/HTTP/Headers/Content-Encoding>

Practically all text compression is done by one of two HTTP content encodings: Gzip<sup>939</sup> and Brotli<sup>940</sup>. Both Brotli and Gzip are supported by virtually all browsers. On the server side, most popular servers<sup>941</sup> like nginx and Apache can be configured to use Brotli and/or Gzip. The configuration is different depending on when the content is generated:

- Static content: this content can be precompressed. The web server can be set up to map the URLs to the appropriate compressed files, e.g. based on the filename extension. For example, CSS and JavaScript are often static content and so can be precompressed to reduce effort for the web server to compress for each request.
- Dynamically generated content: this has to be compressed on the fly for each request by the web server (or a plugin) itself. For example, HTML or JSON can be dynamic content in some cases.

When compressing text with Brotli or Gzip it is possible to select different compression levels. Higher compression levels will result in smaller compressed files, but take a longer time to compress. During decompression, CPU usage tends not to be higher for more heavily compressed files. Rather, files that are compressed with a higher compression level are slightly faster to decode.

Depending on the web server software used, compression needs to be enabled, and the configuration may be separate for precompressed and dynamically compressed content. For Apache<sup>942</sup>, Brotli can be enabled with mod\_brotli<sup>943</sup>, and Gzip with mod\_deflate<sup>944</sup>. For nginx<sup>945</sup> instructions for enabling Brotli<sup>946</sup> and for enabling Gzip<sup>947</sup> are available as well.

## Trends in HTTP compression

The graph below shows the usage share trend of lossless compression from the HTTP Archive metrics over the last 3 years. The usage of Brotli has doubled since 2019, while the usage of Gzip has slightly decreased, and overall the use of HTTP compression is growing on desktop and on mobile.

---

939. <https://tools.ietf.org/html/rfc1952>  
 940. <https://github.com/google/brotli>  
 941. [https://en.wikipedia.org/wiki/HTTP\\_compression#Servers\\_that\\_support\\_HTTP\\_compression](https://en.wikipedia.org/wiki/HTTP_compression#Servers_that_support_HTTP_compression)  
 942. <https://httpd.apache.org/>  
 943. [https://httpd.apache.org/docs/2.4/mod/mod\\_brotli.html](https://httpd.apache.org/docs/2.4/mod/mod_brotli.html)  
 944. [https://httpd.apache.org/docs/2.4/mod/mod\\_deflate.html](https://httpd.apache.org/docs/2.4/mod/mod_deflate.html)  
 945. <https://nginx.org/>  
 946. [https://github.com/google/ngx\\_brotli](https://github.com/google/ngx_brotli)  
 947. [https://nginx.org/en/docs/http/ngx\\_http\\_gzip\\_module.html](https://nginx.org/en/docs/http/ngx_http_gzip_module.html)

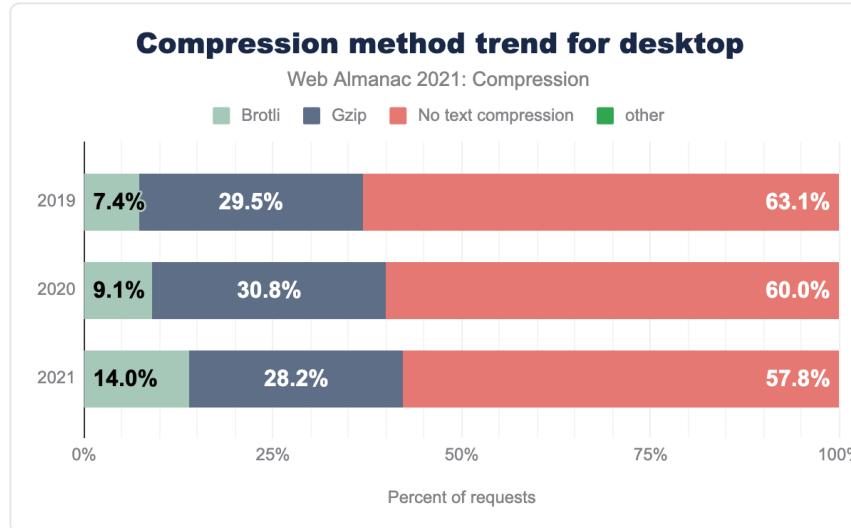


Figure 22.2. Compression method trend for desktop.

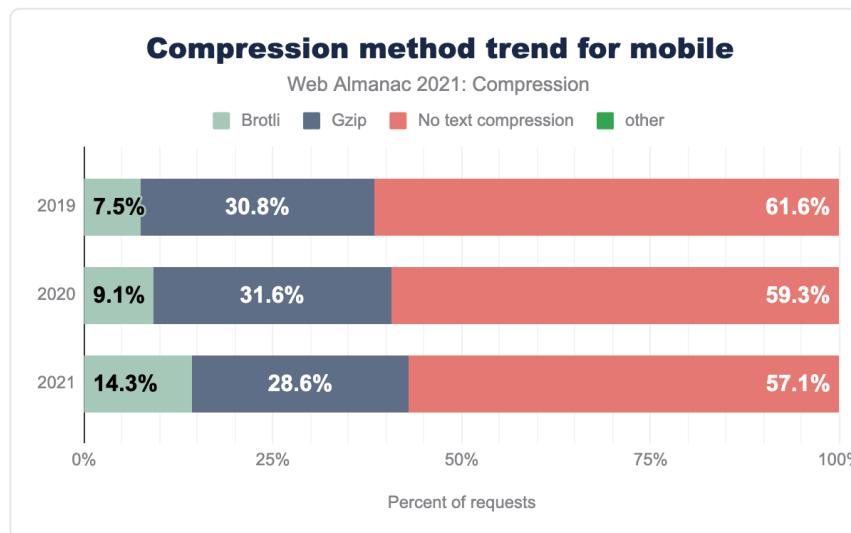


Figure 22.3. Compression method trend for mobile.

Of the resources that are served compressed, the majority are using either Gzip (66%) or Brotli (33%). The other compression algorithms are used infrequently. This split is virtually the same for desktop and mobile.

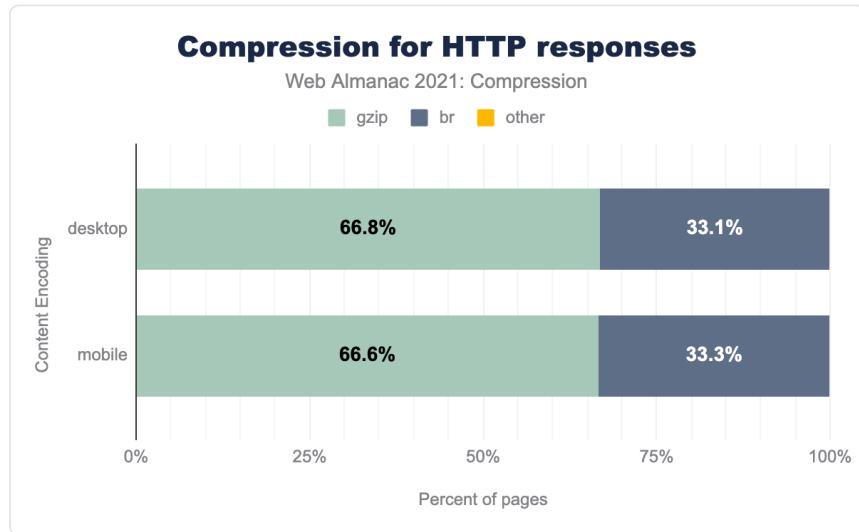


Figure 22.4. Compression algorithm for HTTP responses.

## First-party vs third-party compression

Third Parties have an impact on the user experience of a website. Historically the amount of compression used by first parties compared with third parties was significantly different.

	<b>Desktop</b>	<b>Mobile</b>		
<b>Content-encoding</b>	<b>First-party</b>	<b>Third-party</b>	<b>First-party</b>	<b>Third-party</b>
No text compression	58.0%	57.5%	56.1%	58.3%
Gzip	28.1%	28.4%	29.1%	28.1%
Brotli	13.9%	14.1%	14.9%	13.7%
Deflate	0.0%	0.0%	0.0%	0.0%
Other / Invalid	0.0%	0.0%	0.0%	0.0%

Figure 22.5. First-party versus third-party compression by device type.

From these results we can see that, compared to 2020, first party content has caught up with third party content in the use of compression and they use compression in comparable ways.

Usage of compression and especially Brotli has grown in both categories. Brotli compression has doubled in percentage for first party content compared to a year ago.

## Compression levels

Compression level is a parameter given to the encoder to adjust the amount of effort is applied to find redundancy in the input in order to consequently achieve higher compression density. A higher compression level results in slower compression, but does not substantially affect the decompression speed, even making it slightly faster. For precompressed content, the time needed to compress the data has no effect on the user experience because it can be done beforehand. For dynamic content, the amount time the CPU needs to compress the resource can be traded off to the gain in speed to send the reduced, compressed data over the network.

Brotli encoding allows compression levels from 0 to 11, while Gzip uses levels 1 to 9. Higher levels can be achieved for Gzip as well, with a tool such as Zopfli. This is indicated as `opt` in the graph below.

We used the HTTP Archive `summary_response_bodies` data table to analyze the compression levels currently used on the web. This is estimated by re-compressing the responses with different compression level settings and taking the closest actual size, based on around 14,000 compressed responses that used Brotli, and 11,000 that used Gzip.

When plotting the amount of instances of each level, it shows two peaks for the most commonly used Brotli compression levels, one around compression level 5, and another at the maximum compression level. Usage of compression levels below 4 is rare.

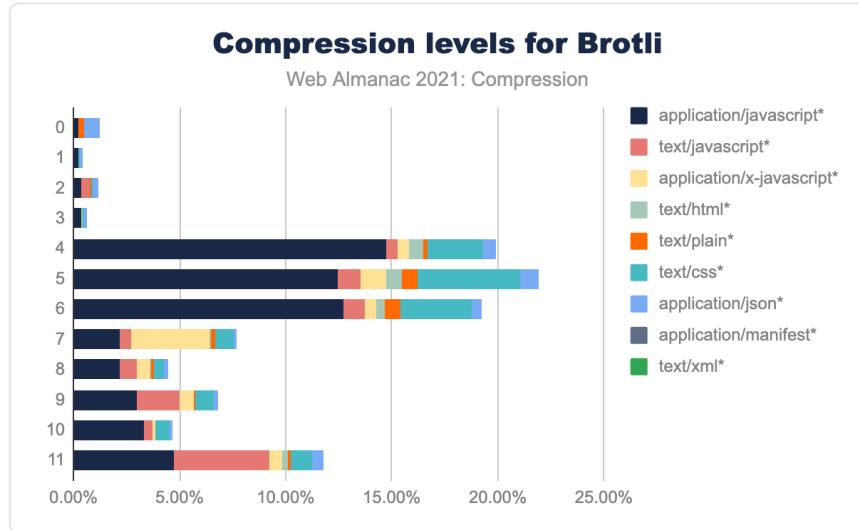


Figure 22.6. Compression levels for Brotli.

Gzip compression is applied largely around compression level 6, extending to level 9. The peak at level 1 might be explained because this is the default compression level of the popular web server nginx<sup>948</sup>. For comparison, Gzip level 9 attempts thousands of redundancy matches, level 6 limits it to about a hundred, while level 1 means limiting redundancy matching to only four candidates and 15% worse compression.

948. <https://nginx.org/>

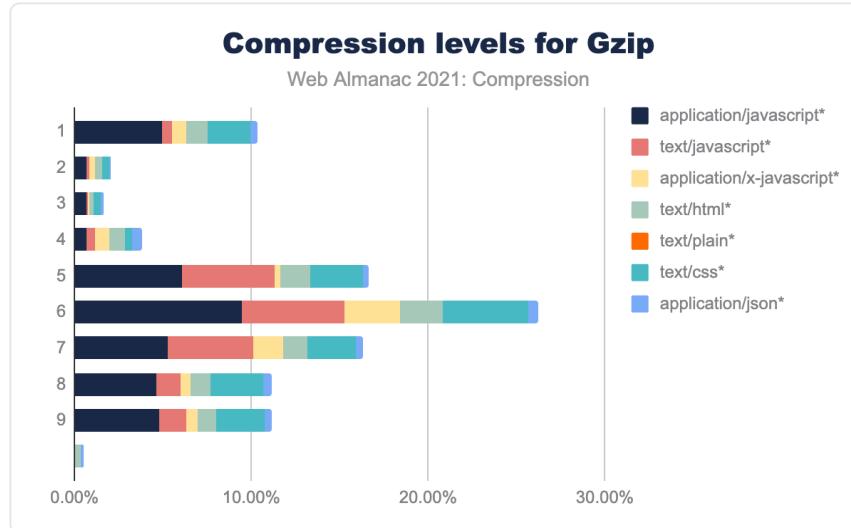


Figure 22.7. Compression levels for Gzip.

The figure breaks down each compression level by content type. JavaScript is the most common content type in almost all cases. For Brotli, the proportion of JavaScript in the highest compression levels is higher than in the lower compression levels, while JSON is more common in the lower compression levels. For Gzip, the distribution of the JavaScript content type is roughly equal at all levels.

## How to analyze compression on sites

To check which content of a website is using HTTP compression, the Firefox Developer Tools<sup>949</sup> or the Chrome DevTools<sup>950</sup> can be used. In the developer tools, open the Network tab and reload your site. A list of responses such as HTML, CSS, JavaScript, fonts and images should appear. To see which ones are compressed, you can check the content encoding in their response header. You can enable a column to easily see this for all responses at once. To do this, right click on the column titles, and in the menu navigate to Response Headers and enable Content-Encoding.

Responses that are Gzip compressed will show “gzip”, while those compressed with Brotli will show “br”. If the value is blank, no HTTP compression is used. For images this is normal, since these resources are already compressed on their own.

949. <https://developer.mozilla.org/docs/Tools>  
950. <https://developers.google.com/web/tools/chrome-devtools>

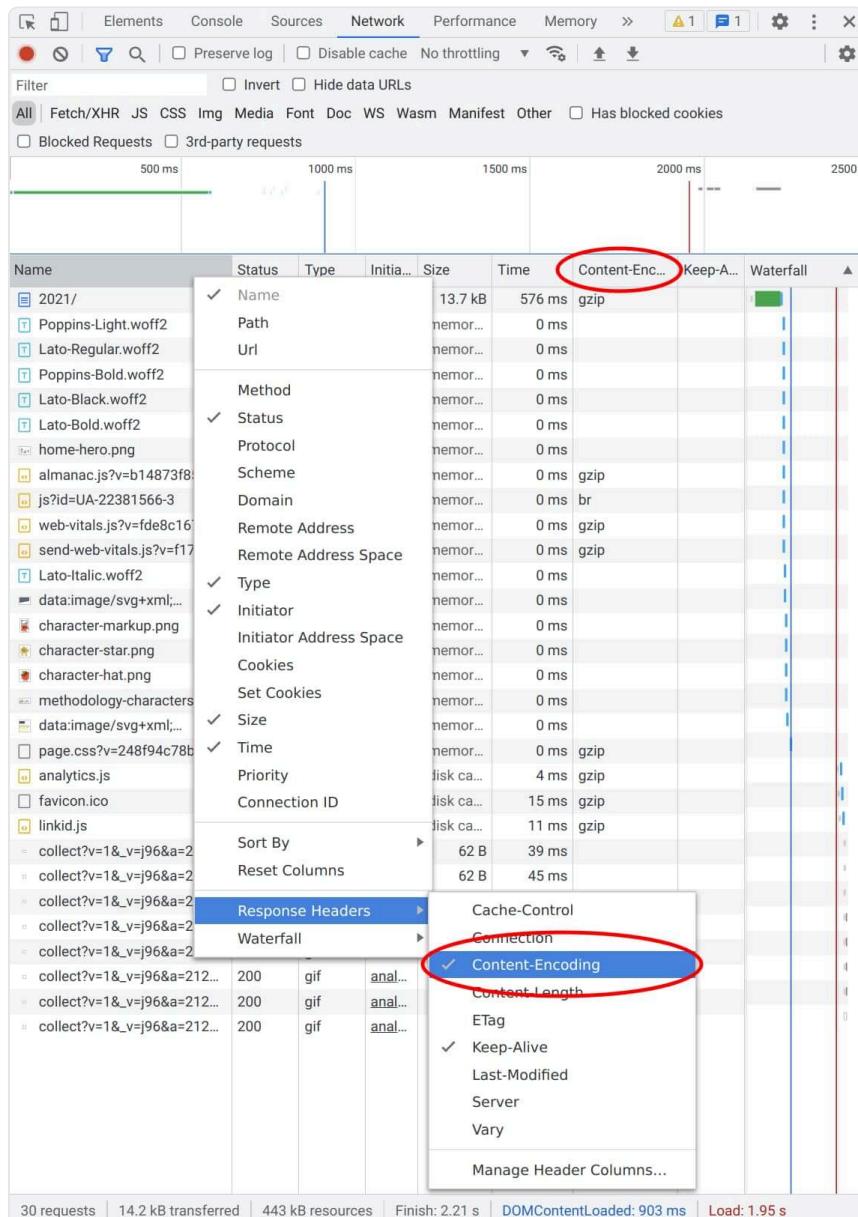


Figure 22.8. Chrome DevTools checking the content-encoding of responses

A different tool that can analyze compression on a site is Google's Lighthouse<sup>951</sup> tool. It runs a series of audits, including the "Enable text compression" audit<sup>952</sup>. This audit attempts to compress resources to check if they reduced by at least 10% and 1,400 bytes. Depending on the score, it can show a compression recommendation in the results, with a list of the resources that can be compressed to benefit a website.

The HTTP Archive runs Lighthouse audits for every mobile page, and from this data we observed that 72% of websites pass this audit. This is 2% less than last year's<sup>953</sup> 74%, which is despite more usage of text compression overall compared to last year, a slight drop.

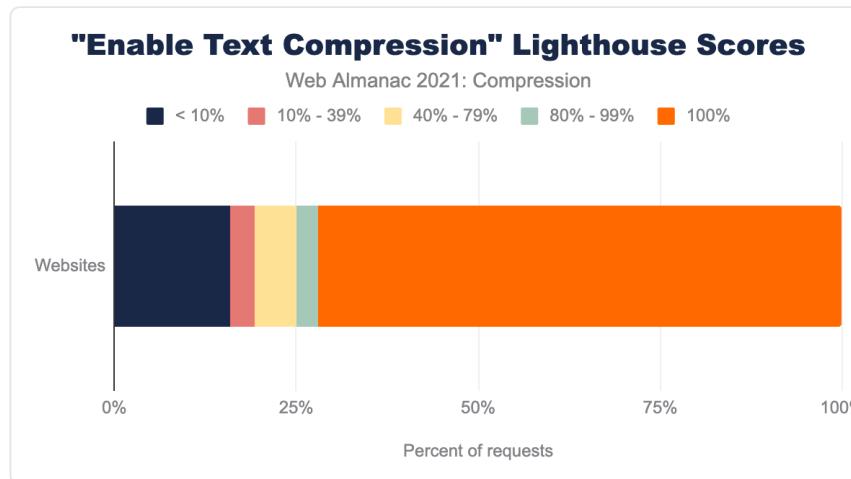


Figure 22.9. Text compression Lighthouse scores.

## How to improve on compression

Before thinking about how to compress content, it is often wise to reduce the content transmitted to begin with. One way of achieving this is to use so-called "minifiers", such as `HTMLMinifier`<sup>954</sup>, `CSSNano`<sup>955</sup>, or `UglifyJS`<sup>956</sup>.

After having the minimal form of the content to transmit, the next step is to ensure compression is enabled. You can verify it is enabled as highlighted in the previous section, and configure your web server if needed.

951. <https://developers.google.com/web/tools/lighthouse>

952. <https://web.dev/uses-text-compression/>

953. <https://almanac.httparchive.org/en/2020/compression#fig-9>

954. <https://github.com/kangax/html-minifier>

955. <https://github.com/ben-eb/cssnano>

956. <https://github.com/mishoo/UglifyJS2>

If using only Gzip compression (also known as Deflate or Zlib), adding support for Brotli can be beneficial. In comparison to Gzip, Brotli compresses to smaller files at the same speed<sup>957</sup> and decompresses at the same speed.

You can choose a well-tuned compression level. What compression level is right for your application might depend on multiple factors, but keep in mind that a more heavily compressed text file does not need more CPU when decoding, so for precompressed assets there's no drawback from the user's perspective to set the compression levels as high as possible. For dynamic compression, we have to make sure that the user doesn't have to wait longer for a more heavily compressed file, taking both the time it takes to compress as well as the potentially decreased transmission time into account. This difference is borne out when looking at compression level recommendations for both methods.

When using Gzip compression for precompressed resources, consider using Zopfli<sup>958</sup>, which generates smaller Gzip compatible files. Zopfli uses an iterative approach to find an very compact parsing, leading to 3-8% denser output, but taking substantially longer to compute, whereas Gzip uses a more straightforward but less effective approach. See this comparison between multiple compressors<sup>959</sup>, and this comparison between Gzip and Zopfli<sup>960</sup> that takes into account different compression levels for Gzip.

	<b>Brotli</b>	<b>Gzip</b>
Precompressed	11	9 or Zopfli
Dynamically compressed	5	6

Figure 22.10. Recommended compression levels to use.

Improving the default settings on web server software would provide significant improvements to those who are not able to invest time into web performance, especially Gzip quality level 1 seems to be an outlier and would benefit from a default of 6, which compresses 15% better on the HTTP Archive `summary_response_bodies` data. Enabling Brotli by default instead of Gzip for user agents that support it would also provide a significant benefit.

## Conclusion

The analysis of compression levels used on 28,000 HTTP responses reveals that about 0.5% of Gzip-compressed content uses more advanced compressors such as Zopfli, while a similar

957. <https://quixdb.github.io/squash-benchmark/>

958. <https://en.wikipedia.org/wiki/Zopfli>

959. <https://cran.r-project.org/web/packages/brotli/vignettes/brotli-2015-09-22.pdf>

960. <https://blog.codinghorror.com/zopfli-optimization-literally-free-bandwidth/>

“optimal parsing” approach is used for 17% of Brotli-compressed content. This indicates that when more efficient methods are available, even if slower, a significant number of users will deploy these methods for their static content.

Usage of HTTP compression continues to grow, and especially Brotli has increased significantly compared to the previous year’s chapter<sup>961</sup>. The number of HTTP responses using any text compression increased by 2%, while Brotli increased by over 4%. Despite the increase, we still see opportunities to use more HTTP compression by tweaking the compression settings of servers. You can benefit from taking a closer look at your own website’s responses and your server configuration. Where compression is not used, you may consider enabling it, and where it is used you may consider tweaking the compression methods towards higher compression levels, both for dynamic content such as HTML generated on the fly, and static content. Changing the default compression settings in popular HTTP servers could have a great impact for users.

## Authors



### Lode Vandevenne

lvandeve

Lode Vandevenne works at Google Switzerland as a software engineer and has contributed to compression projects including Zopfli, Brotli and the JPEG XL image format.



### Moritz Firsching

mo271

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.

<sup>961.</sup> <https://almanac.httparchive.org/en/2020/compression>



## Jyrki Alakuijala

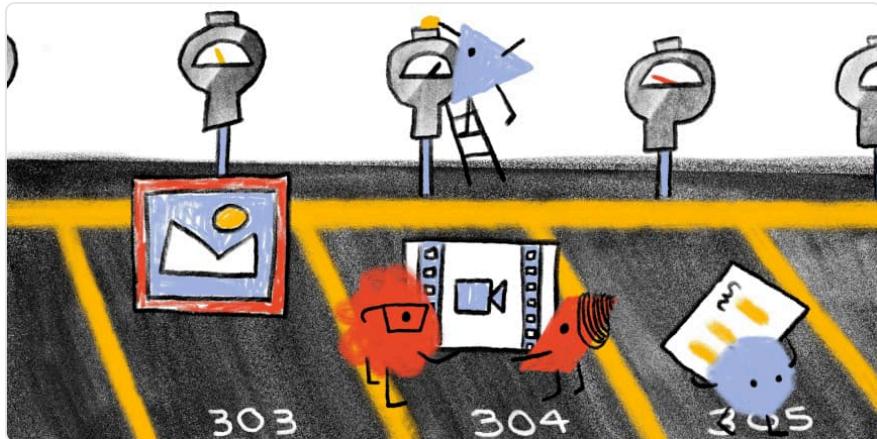
🐦 @jyzg ⚡ jyrkialakuijala

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, Giffeli, 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.



# Part IV Chapter 23

# Caching



*Written by Leonardo Zizzamia and Jessica Nicolet*

*Reviewed by Wilhelm Willie and Rory Hewitt*

*Analyzed by Rick Viscomi*

*Edited by Barry Pollard*

## Introduction

Over the last two decades, the way we experience web applications has changed, giving us richer and more interactive content. Unfortunately, this content comes with a cost in both data storage and bandwidth. Most of the time, this makes it harder for many of us to fully experience a web product when the network we use is degraded, or our device doesn't have enough space. Caching is both a solution to and the cause of some of these problems. Learning to navigate the multitude of choices will enable you to build not only for high-end devices but also for the next billion users that access your product from low-end devices.

Caching is a technique that enables the reuse of previously downloaded content, from simple static assets like JavaScript, CSS files or basic string values to more complex JSON API responses.

At its core, caching avoids making specific HTTP requests and allows an application to feel more responsive and reliable to the user. Each request is usually cached in two main places:

- **Content Delivery Networks (CDNs)** is usually a third-party company with the primary goal of replicating your data as closely as possible to where the user is accessing the application. Most CDNs have some default behavior, but mainly you can give them instructions on how to cache by using headers.
- **Browsers** will either respect the HTTP headers you defined to optimize the experience, or apply some internal defaults. On top of that, browsers provide access to additional manual caching strategies including storing simple strings in *cookies*, complex API responses in *IndexedDB*, or entire resources using *CacheStorage* with a *service worker*.

In this chapter, we will mostly focus on the HTTP headers used between the browser and the CDN, briefly mentioning service worker caching strategies.

## CDN cache adoption

A Content Delivery Network (CDN) is a group of servers spread out over several locations that usually store copies of data. This allows servers to fulfill requests based on the server closest to the end-user.

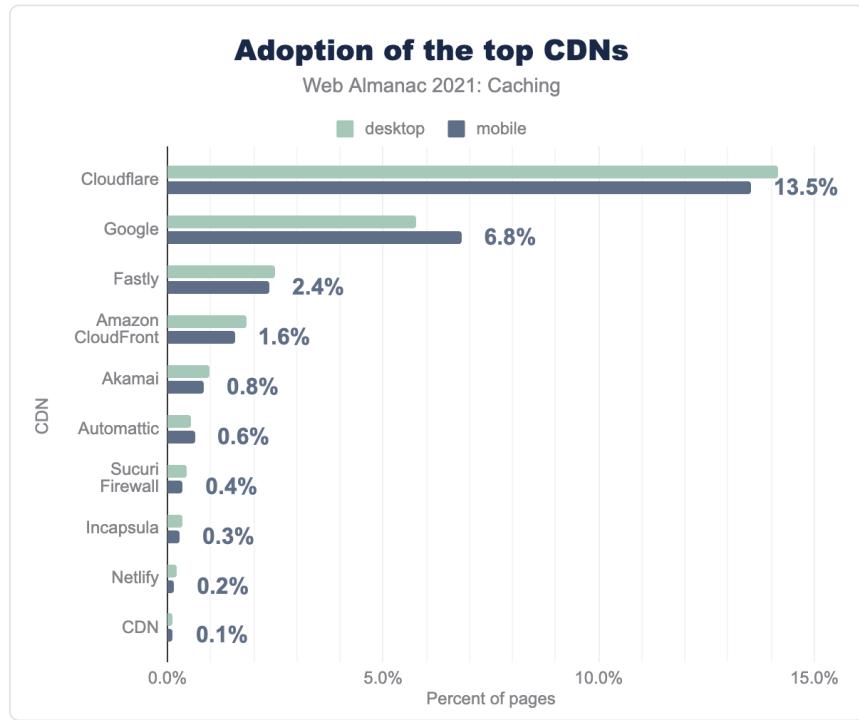


Figure 23.1. Adoption of the top CDNs.

Across the web in 2021, the most popular CDN used for Desktop was Cloudflare with 14% of total pages, followed by Google with 6%. While Cloudflare and Google are the most popular, a large variety of solutions remain available beyond these two including Fastly, Amazon CloudFront, Akamai, and many others.

## Service worker adoption

The adoption of *service workers* has continued to steadily increase.

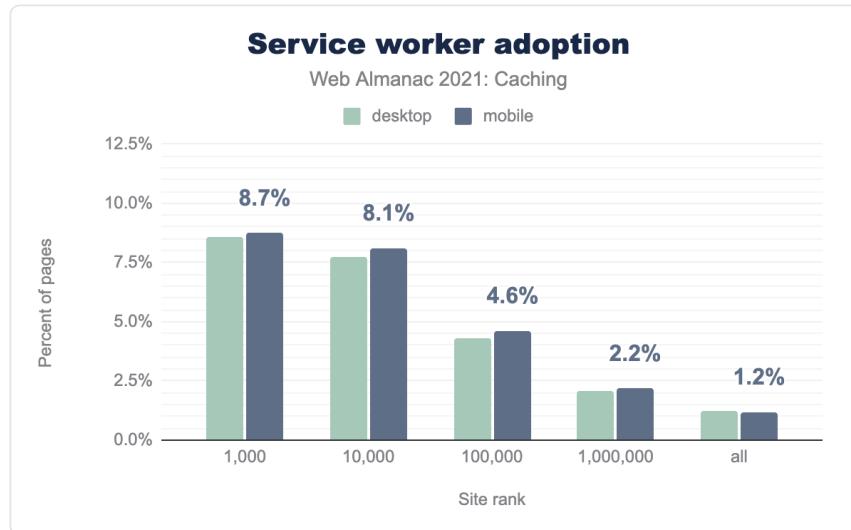


Figure 23.2. Service worker adoption.

While just over 1% of pages registered a service worker, nearly 9% of pages ranked in the top 1,000 most visited domains registered one.

This higher adoption of service workers, particularly in the top 1,000 pages, could be related to the world-wide trend towards remote-first and by association, mobile-friendly. As our reliance on working and living in one place throughout the entire year shifts, we need our devices to work even harder and smarter to keep up with us. Service workers are a tool that can improve performance when the user is dealing with unreliable networks or low-end devices.

The primary way to cache resources within a service worker is by using the *CacheStorage API*. This allows a developer to create a custom cache strategy for any requests passing through the worker; some well-known ones are *stale-while-revalidate*, *Cache Falling Back to Network*, *Network Falling Back to Cache*, and *Cache Only*. In recent years it has become even easier to adopt those strategies thanks to the increased popularity of Workbox<sup>962</sup>, which helps you decide what cache you want to plug and play.

Service workers, and Workbox, are discussed in more detail in the PWA chapter.

<sup>962</sup> <https://developers.google.com/web/tools/workbox/modules/workbox-strategies>

## Caching headers adoption

With both a CDN and the Browser, HTTP headers are the primary tool a developer must master to properly cache resources. Headers are simply instructions read from the HTTP request or response, and some of them help control the cache strategy used.

The caching-related headers, or the absence of them, tell the browser or CDN three essential 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 ensure that my cached version is still fresh?

Headers are meant to be used either alone or together. To determine if the content is **cacheable** and **fresh**, we have:

- `Expires` specifies an explicit expiration date and time (i.e., when precisely 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 generated).

When both are specified, `Cache-Control` takes precedence.

## Usage of Cache-Control and Expires headers

Web Almanac 2021: Caching

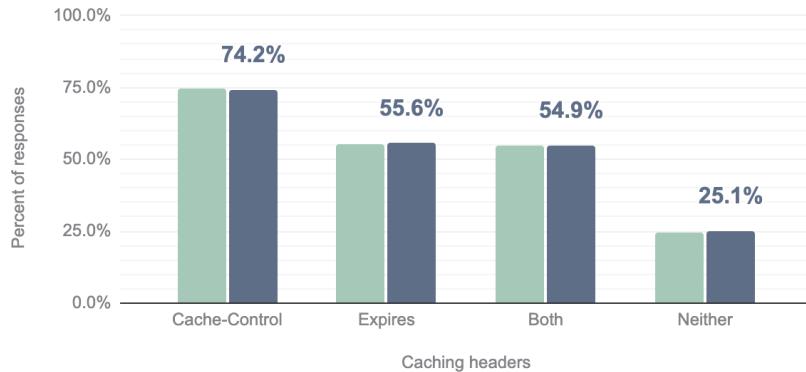



Figure 23.3. Percent of responses that set `Cache-Control` and `Expires` headers.

Usage of the `Cache-Control` header has increased steadily since 2019. 74.2% of responses on mobile requests included the `Cache-Control` header, while 74.8% of responses on desktop requests utilized the header.

Since 2020, the usage of this specific header increased by 0.71% for mobile and by 1.13% for desktop. But on mobile, we still have 25.1% of requests using neither `Cache-Control` nor `Expires` headers. This leads us to believe there has been an increase in awareness in the community around proper usage of `Cache-Control`, but we still have a long way to go to full adoption of these headers.

To validate the content, we have:

- `Last-Modified` indicates when the object was last changed. Its value is a date timestamp.
- `ETag` (Entity Tag) provides a unique identifier for the content as a quoted string. It can take any format the server chooses. It is typically a hash of the file contents, but it can be a timestamp or a simple string.

When both are specified, `ETag` takes precedence.



Figure 23.4. Percent of responses that set `Last-Modified` and `ETag` headers.

Comparing 2020 and 2021, we notice a recurring trend from past years with the `ETag` becoming slightly more popular each year, and `Last-Modified` being used 1.5% less. What we should probably keep an eye on next year is a new trend of 1.4% more responses using neither `ETag` nor `Last-Modified` headers, as this could imply a challenge in the community understanding the value of these headers.

### `Cache-Control` directives

When using 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 set in any order, although some clash with one another (e.g., `public` and `private`). In addition, some directives take a value, such as `max-age`.

Below is a table showing the most common `Cache-Control` directives:

Directive	Description
max-age	Indicates the number of seconds that a resource can be cached for relative to the current time. For example, max-age=86400.
public	Indicates that any cache can store the response, including the browser and the CDN. This is assumed by default.
no-cache	A cached resource must be revalidated before its use, via a conditional request, even if it is not marked as stale.
must-revalidate	A stale cached entry must be revalidated before its use, via a conditional request.
no-store	Indicates that the response must not be cached.
private	The response is intended for a specific user and should not be stored by shared caches such as CDNs.
immutable	Indicates that the cached entry will never change during its TTL, and that revalidation is not necessary.

## Usage of Cache-Control directives

Web Almanac 2021: Caching

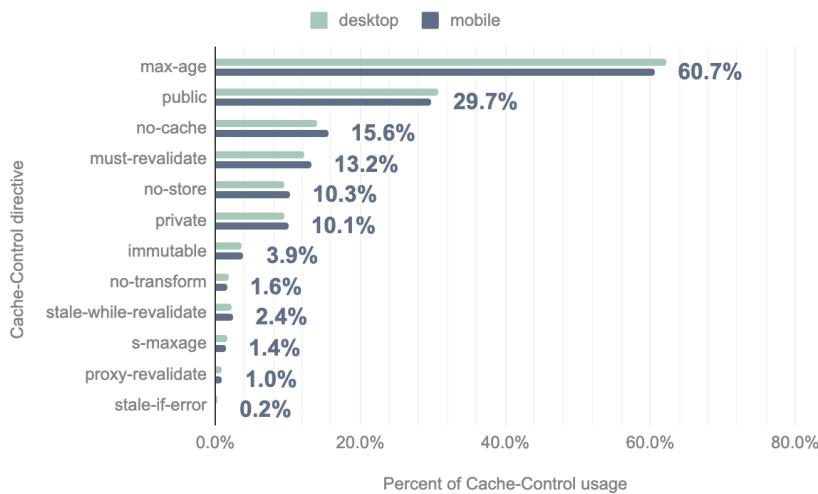


Figure 23.5. Usage of `Cache-Control` directives.

The `max-age` directive is the most commonly found with 62.2% of desktop requests including a `Cache-Control` response header with this directive.

Compared to 2020, `max-age` adoption increased by 2% on desktop, along with most of the top seven directives in the above chart.

The `immutable` directive is relatively new and can significantly improve cacheability for certain types of requests. However, it is still only supported by a few browsers, and we see most requests coming from host networks like Wix with 16.4%, Facebook with 8.6%, Tawk with 2.8%, and Shopify with 2.4%.

The most misused `Cache-Control` directive continues to be `set-cookie`, used for 0.07% of total directives for desktop and 0.08% for mobile. However, we are pleased to see a meaningful 0.16% reduction of usage from 2020.

When we take a look when `no-cache`, `max-age=0` and `no-store` are used together, we also see a growing trend year after year, in which `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. Driving more awareness around using these directives, for example during larger conferences, could help avoid accidentally wasted bytes.

## 51 trillion years

Figure 23.6. Largest recorded value for `max-age`

Fun fact: The most common `max-age` value is 30 days, and the largest value is 51 trillion years.

### 304 Not Modified status

When it comes to size, `304 Not Modified` responses are much smaller than `200 OK` responses, so it follows that page performance can be sped up by only delivering the necessary size of data. This is where correctly using conditional requests comes in. Revalidation, and therefore data savings, can be done by using either an `ETag` or `Last-Modified` header.

The `Last-Modified` response header works in conjunction with the `If-Modified-Since` request header to let the browser know if any changes have been made to the requested file.

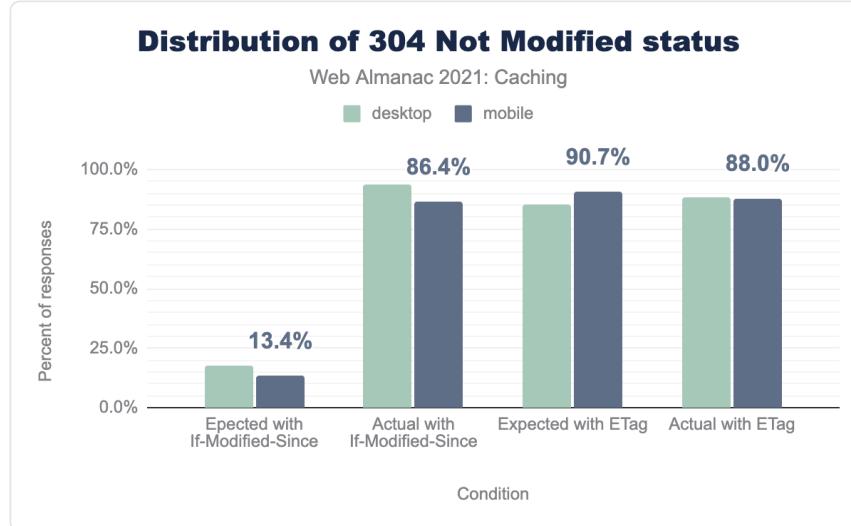


Figure 23.7. HTTP 304 response rate by caching strategy.

We saw the distribution of 304 responses increase by 7.7% for `If-Modified-Since` between 2020 and 2021. This shows that the community is capitalizing on these data savings.

## Validity of date strings

The three main HTTP headers used to represent timestamps, `Date`, `Last-Modified`, and `Expires` all use a date formatted string. The `Date` HTTP response header is almost always generated automatically by the web server, meaning that invalid values are extremely rare. Still, in the event that the date is set incorrectly it can affect cacheability on the response on which it is served.

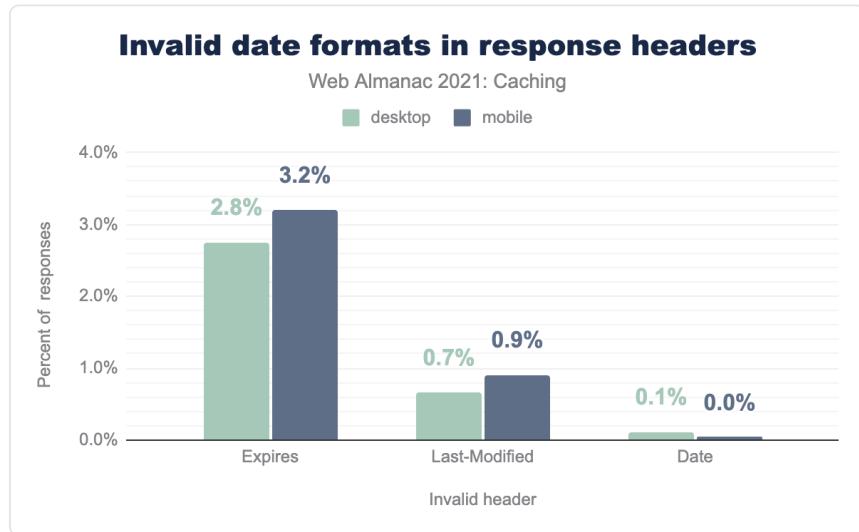


Figure 23.8. Percent of responses with invalid date formats.

Between 2020 and 2021, the percent using invalid `Date` improved by 0.5% but worsened for `Last-Modified` and `Expires` showing that it was related to how the date was set on caching.

This shows us that automation of the date-based headers could benefit from further attention.

## Vary

An essential step in caching a resource is understanding if it was previously cached. The browser typically uses the URL as the cache key. At the same time, requests for the same URLs but with different `Accept-Encoding` will result in different responses and so could be cached incorrectly. That's why we use the `Vary` header to instruct the browser to add a value of one or more headers to the cache key.



Figure 23.9. Usage of `Vary` directives.

The most popular `Vary` header is `Accept-Encoding` with 90.3% usage, followed by `User-Agent` with 10.9%, `Origin` with 10.1%, and `Accept` with 4.8%.

We saw a 1.5% decrease in use of `Accept-Encoding` from 2020.

# 46.3%

Figure 23.10. Percent of mobile responses that set the `Vary` header.

It's important to point out that only 46.25% of total requests audited use the `Vary` header, but when compared to 2020, we see an overall increase by 2.85%.

# 83.4%

Figure 23.11. Percent of mobile responses with the `Vary` header that also set `Cache-Control`.

Of the requests using the `Vary` header, 83.4% also have the `Cache-control`. This shows us a 2.1% improvement from 2020.

## Setting cookies on cacheable responses

In the 2020 Caching chapter, we were reminded to be aware of using `set-cookie` with cacheable responses because only 4.9% of responses used the `private` directive, putting a user's private data at risk of being accidentally served to a different user via a CDN.

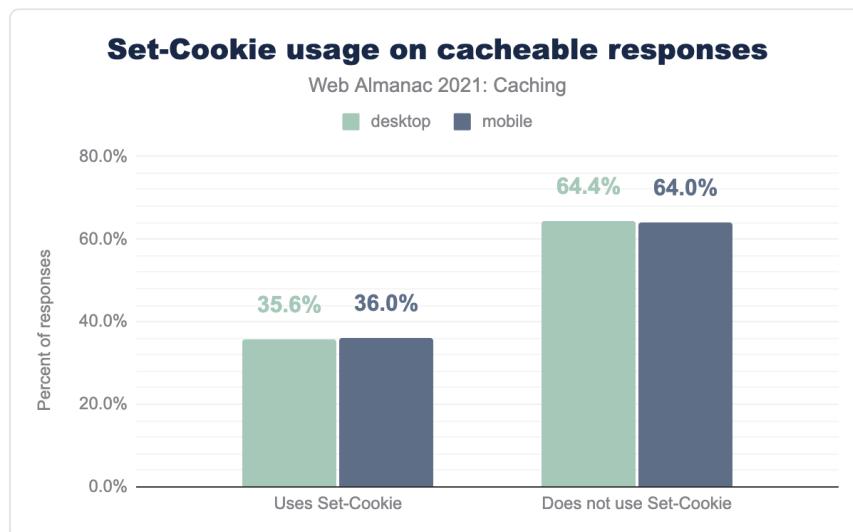


Figure 23.12. Percent of cacheable responses that use `Set-Cookie`.

In 2021, we see an increase in awareness regarding `set-cookie` and caching coexisting. While still only 5% of web pages are using the `private` directive with `set-cookie`, the total number of cacheable `set-cookie` responses decreased by 4.41%.

## What type of content are we caching?



Figure 23.13. The percent of requests that use caching strategies by resource type.

Font, CSS, and audio files are over 99% cacheable, with almost 100% of pages currently caching fonts. This is likely due to their static nature, making them prime choices for caching.

However, some of our most commonly used resources are non-cacheable, likely due to their dynamic nature. Notably, HTML saw some of the highest percentage of non-cacheable resources at 23.4%, followed closely by images with 10.1%.

When we compare the mobile data between 2020 and 2021, we notice a 5.1% increase in cacheable HTML. This tells us we may be moving towards better usage of our CDNs to cache HTML pages, like those generated by Server-Side Rendered applications. Pages are typically generated by SSR if the content of a particular web page doesn't change frequently. The URL can potentially serve the same HTML for weeks or even months, making that content highly cacheable.

Type	Desktop	Mobile
Text	0.2	0.2
XML	1	1
Other	1	1
Video	4	8
HTML	3	14
Audio	0.2	30
CSS	30	30
Image	30	30
Script	30	30
Font	365	365

Figure 23.14. Median TTL (in days)

Taking a look at the median *Time To Live* (TTL) across all resource types, we see that even if we cache a similar percentage in total, there is a much longer cache for mobile, particularly for HTML, audio and video.

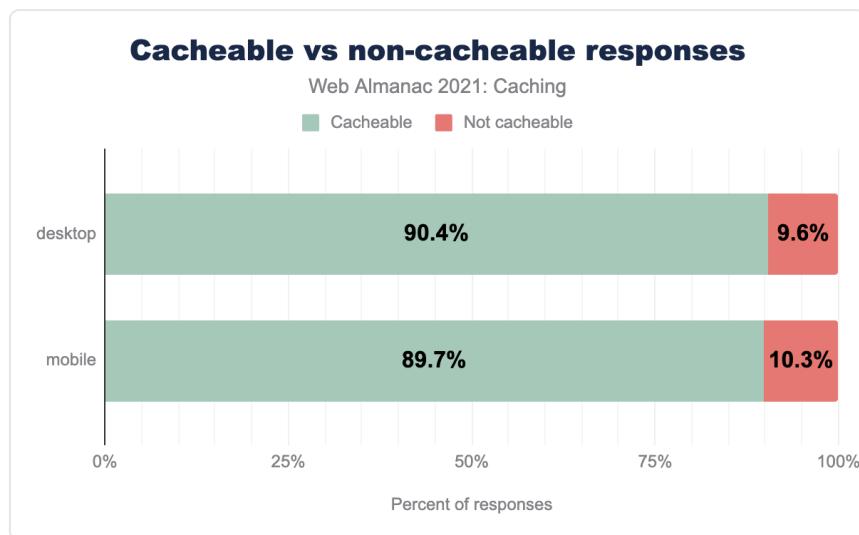


Figure 23.15. Percent of cacheable vs non-cacheable responses.

That said, even as we continue to optimize for the mobile experience, it's interesting to note that the potential amount of cacheable desktop resources remains slightly higher than those for mobile.

## How do cache TTLs compare to resource age?



Figure 23.16. Distribution of first-party resource age by content type (mobile only).

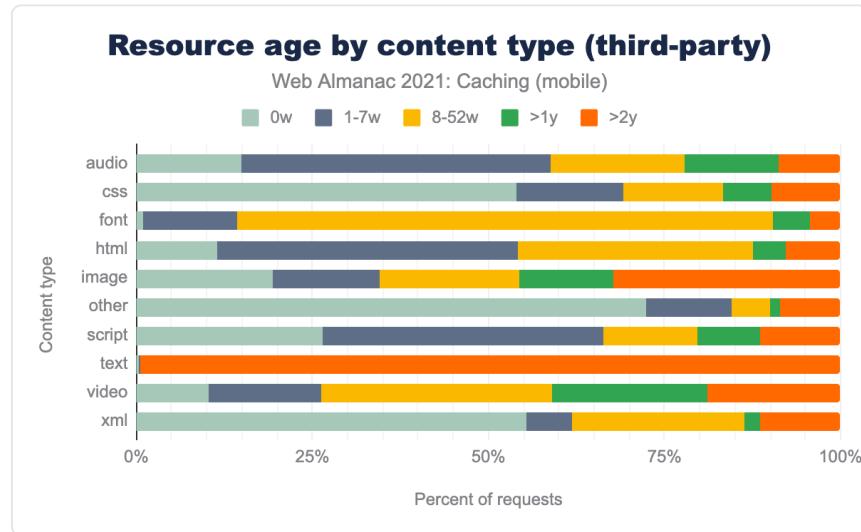


Figure 23.17. Distribution of third-party resource age by content type (mobile only).

We see that images and videos maintained the same average age whether from first or third-party sources. Images consistently had a resource age of 2 years, while most video resources were between 8-52 weeks old.

Breaking down the other types of content, we discovered fonts for third-parties are cached the most between 8-52 weeks at 72.4%. However, for first-party the largest resource age group is evenly split between 8-52 weeks and over 2 years- quite a large variance. We see similar results for audio and scripts where the majority of first-party are between 8-52 weeks old while for third-party they are between 1-7 weeks.

Audio was the most highly cached resource across both first and third parties. However, the resource age varied greatly between first-party (averaging 8-52 weeks) and third-party, at only 1-7 weeks. Audio resources in first-party situations tend to be updated less frequently (why?), so third parties may be capitalizing on a caching opportunity by offering fresher resources.

The largest group of cached first-party CSS (32.2%) tended to be 8-52 weeks old, while the largest group for 3rd parties was less than a week with 51.8% of resources cached for that duration.

Finally, HTML has the largest first-party group served with less than a week with 42.7% and third-party's largest group is between 1-7 weeks with 43.1%.

Considerations after reviewing this data:

- The freshest content for first-party is HTML while for third-party it is CSS.
- The stalest content for both first and third-party is images.

This data shows us that first parties have prioritized refreshing HTML content, which usually holds the link to JS and CSS files, while third-party providers that are mostly CSS and script-driven, like browser extensions, have prioritized keeping their CSS up to date. When we consider the origins behind first parties vs. third parties, it follows that the way content is delivered may be more important to third parties than the actual content, thus making their presentation and optimization of it, all the more important.

Mobile resources with a cache TTL that was considered too short compared to its content age have seen an improvement since 2020. This data is exciting because it hints at the community's growing understanding of appropriately relative caching.

# 54%

*Figure 23.18. 54% of mobile resources are older than their TTL*

While a cache TTL that is too long may serve stale content, there is no benefit for the end user if it is too short. The connection between cache TTL and content age is slowly closing this gap, moving from 60.2% in 2020 to 54.3% in 2021. The more attentive we can be towards to content age (i.e. how often we revamp a page's HTML, CSS etc.), the more accurately we can set cache limits.

Developers are getting better at setting the cache duration more accurately to the content age, resulting in more responsible, and therefore more effective, caching.

<b>Client</b>	<b>First-party</b>	<b>Third-party</b>	<b>Overall</b>
Desktop	59.5%	46.2%	54.3%
Mobile	60.1%	44.7%	54.3%

*Figure 23.19. Percent of requests with short TTLs.*

When we split the data between first and third-party providers, the largest improvements come from 3rd parties where we have a 13.2% improvement. It is highly encouraging to see companies around the world building products for developers that are optimizing caching. It's possible that the developer community's increased attention towards improving performance has encouraged and even incentivized 3rd parties to optimize their caching strategies.

However, the challenge remains for how first parties can effectively improve over the coming years.

## Identifying caching opportunities



Figure 23.20. Distribution of Lighthouse caching TTL scores.

Based on the Lighthouse *caching TTL* score, we have seen an improvement in pages ranked with a perfect score of 100 increase from 3.3% in 2020 to 4.4% in 2021.

The score reflects whether the pages can benefit from additional caching policy improvements. Even though we are excited to see 31% of pages scoring above the 50th percentile score, a large potential for improvement exists for the 52% of pages that are ranking below the 25th percentile.

This makes us consider that even though web pages have some level of caching, the way the policies are used is outdated and not optimized to the latest state of their products.

## Distribution of potential byte savings from caching

Web Almanac 2021: Caching (mobile)



Figure 23.21. Distribution of potential byte savings from caching.

Based on Lighthouse wasted bytes audit from 2020 to 2021, there was a 3.28% improvement in wasted bytes across all audited pages on repeated views. This lowers the percentage of pages that waste 1 MB from 42.8% to 39.5%, showing a considerable trend from the community towards building products that are less costly for international users with paid internet data plans.

The current percentage of pages audited that have 0 wasted bytes is still relatively low at 1.34%. In the coming years, we're looking forward to seeing an increase in that percentage as the community continues to focus on optimizing web performance.

## Conclusion

The late, great Phil Karlton<sup>963</sup> famously said, “There are only two hard things in Computer Science: cache invalidation and naming things.”, and in all honesty I have always wondered why caching is so hard. My take is that to do caching well, you need two key ingredients: to keep it simple and to understand all potential edge cases.

Unfortunately, when we try to make the cache too clever, we can end up caching the wrong things or, worse, caching too much. On a similar note, understanding all the edge cases requires extensive research, testing, and slow incremental improvements. Even with that, you have to

<sup>963</sup>. <https://www.karlton.org/karlton/>

hope that an old browser will not throw you under the bus. But the reason we still chase great caching strategies is that the ultimate reward is very high, with a significant reduction in round-trip requests, high savings for your server, less data required from your users, and ultimately a better user experience.

No matter the case, make sure to have a playbook for how to best use caching:

- Prioritize caching work at an early stage of the development cycle, *and* after a product is shipped
- Write end-to-end tests to recreate major edge cases
- Regularly audit the site and update cache rules that might be outdated or missing

Ultimately, caching can be made less complex if we spread the knowledge by mentoring our peers and writing good documentation that is simple to understand. Caching is not something that should only be mastered by a few. Our goal is to move towards it being common knowledge across an entire company. Because at the end of the day, what we really want to focus on is building easy and frictionless experiences for our users.

## Authors



### 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 Coinbase<sup>964</sup>, leading initiatives that enable product engineers to ship the highest quality applications in the world at scale. He curates the NGRome Conference<sup>965</sup>. Leo also maintains the Perfume.js<sup>966</sup> library, which helps companies prioritize roadmaps and make better business decisions through performance analytics.

964. <https://www.coinbase.com/>  
965. <https://ngrome.io>  
966. <https://github.com/Zizzamia/perfume.js>



## Jessica Nicolet

Twitter: [@jessica\\_nicolet](https://twitter.com/jessica_nicolet) | GitHub: [@jessnicolet](https://github.com/jessnicolet) | Website: <https://www.jessicanicolet.com/>

Jessica began her career as an opera singer and has been in the classical music industry for the past 10 years. In early 2020 and due to the pandemic, she decided to start a new career in Tech, specifically Web Development. She has always enjoyed writing and telling stories both on stage and off and published a series of three articles<sup>967</sup> on Medium documenting her experience transitioning to this new field. She is currently looking for a full-time position in Technical Writing.

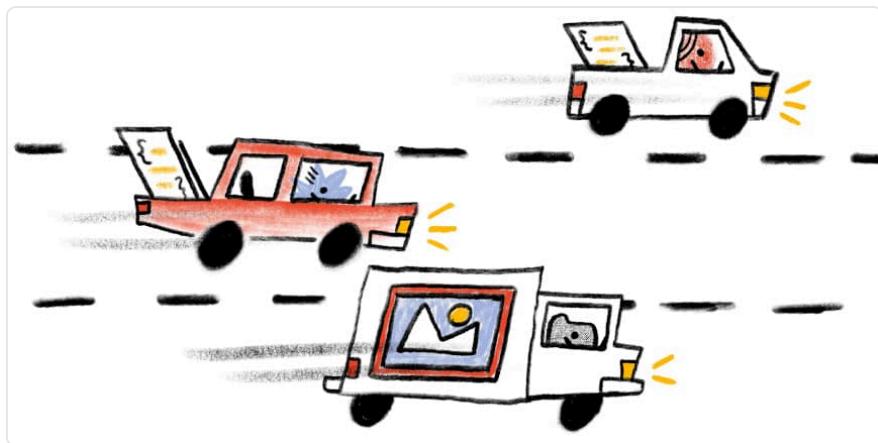
---

---

<sup>967</sup>. <https://jessicanicolet.medium.com/>

# Part IV Chapter 24

# HTTP



*Written by Dominic Lovell*

*Reviewed by Barry Pollard and Robin Marx*

*Analyzed by Barry Pollard*

*Edited by Shaina Hantsis*

## Introduction

The HTTP protocol is one of key parts of the web. HTTP itself was unchanged for nearly two decades after HTTP/1.1 was introduced in 1997. It wasn't until 2015 with the introduction of HTTP/2, that saw a major design change to the way HTTP was implemented. HTTP/2 was designed to introduce changes primarily at the transport level of the protocol. These protocol changes, while significant in how they worked, still allowed for backward compatibility between versions.

This year we again take a closer look at HTTP/2, discussing some of its major features. We then look at some of the benefits of HTTP/2, and why it has been adopted heavily across the web performance community. While HTTP/2 aimed at solving many problems with HTTP, including connection limits, better header compression, and binary support which allowed for better payload encapsulation, not all features put forward were successful in their design.

After several years of HTTP/2 in the wild, some of the intentions of HTTP/2 are still to be

realized. For example, last year we put forward the question of whether we say goodbye to HTTP/2 push. This year we aim to answer this question with more confidence by looking at the 2021 data. As these shortcomings came to light, they have been addressed or omitted from the next iteration of HTTP: HTTP/3.

Increased support for HTTP/3 over the past year has allowed for introspection on HTTP/3's adoption on the web. This chapter takes a closer look at some of the core features of HTTP/3 and the benefits of each of these. We also examine the major vendors who are supporting HTTP/3 evolution, as well as some of the ongoing critiques of HTTP/3.

Some of the data points the Web Almanac aims to answer across the HTTP chapter include the adoption across HTTP versions, support from the key software vendors and CDN companies, and how this distribution between first and third parties influences adoption. We also take a look at usage across the top ranked sites across the web, including metrics on HTTP attributes such as connections, server push and response data size.

These data points provide a snapshot for 2021 on the HTTP usage across the web and how the protocol is evolving across its major versions. They then provide insight into the adoption of major features in the coming years.

## Evolution of HTTP

It's been six years since the Internet Engineering Task Force (IETF)<sup>968</sup> introduced us to HTTP/2<sup>969</sup>, and it's worth understanding how we got to HTTP/2 in the first place. Thirty years ago (in 1991) we were first introduced to HTTP 0.9. HTTP has come a long way since 0.9, which was limited in capabilities. 0.9 was used for one-line protocol transfers, which only supported the GET method, and had no support for headers nor status codes. Responses were only provided in hypertext. Five years later, this was enhanced with HTTP/1.0. The 1.0 version contains most of the protocol we know now, including response headers, status codes, and the `GET`, `HEAD` and `POST` methods.

A problem not addressed in 1.0 was that the connection was terminated immediately after the response was received. This meant each request was required to open a new connection, perform TCP handshakes, and close the connection after the data was received. This major inefficiency saw HTTP/1.1 introduced only a year later in 1997, which allowed for persistent connections to be made, which can be reused once opened. This version served its purpose for 18 years, without any changes introduced until 2015. During this time Google experimented with SPDY<sup>970</sup>—a complete reimaging of how HTTP messages were sent. This was eventually formalized into HTTP/2.

968. <https://www.ietf.org/>

969. <https://datatracker.ietf.org/doc/html/rfc7540>

970. <https://en.wikipedia.org/wiki/SPDY>

HTTP/2 aimed to address many of the problems web developers were facing when trying to achieve increased performance. Complicated processes such as domain sharding, asset spriteing, and concatenating files were necessary to work around inefficiencies in HTTP/1.1. By introducing resource multiplexing, prioritization, and header compression, HTTP/2 was designed to provide network optimization at the protocol level. As well as addressing the known performance problems, HTTP/2 introduced new potential performance optimizations with features such as *HTTP/2 push*, where the server could preemptively send content to the client before the client would be aware of the asset.

## Adoption of HTTP/2

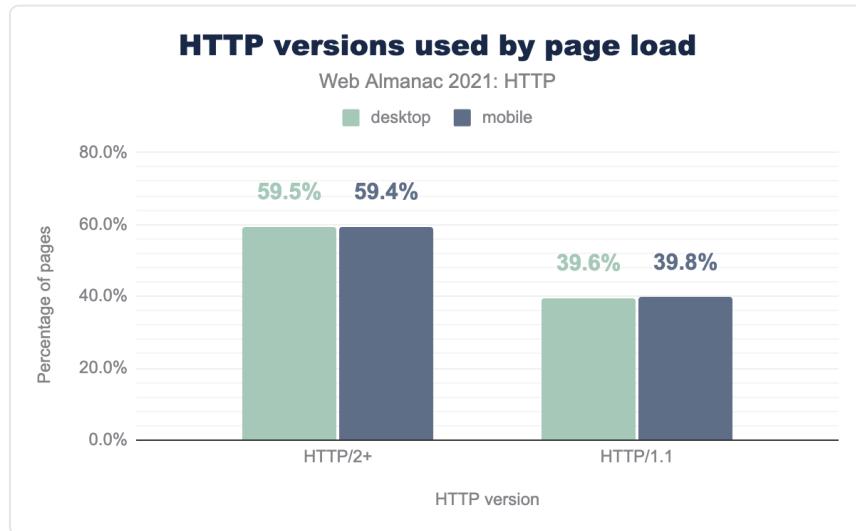


Figure 24.1. HTTP versions used by page load.

In the thirty years since HTTP version 0.9, there has been a shift in the protocol's adoption. With over 6 million web pages analyzed, the HTTP Archive found only a single instance of HTTP 0.9 being used for the initial page request, only a couple of thousand pages still using 1.0. Almost 40% of pages were still using version 1.1 however, with the remaining 60% using HTTP/2 or above. HTTP/2 adoption is thus up 10% since the same analysis was performed in 2020.

*Note: Due to the way HTTP/3 works, as we will discuss below, and how our crawl works with a fresh instance each time, HTTP/3 is unlikely to be used for the initial page request, or even subsequent requests. Therefore, we report some statistics in this chapter as "HTTP/2+" to indicate HTTP/2 or HTTP/3 might be used in the real world. We will investigate how much HTTP/3 is actually supported*

(even if not used in our crawl) later in the chapter.

## Adoption by request

The initial page request is supplemented by many other requests, often served by third parties, which may have different, often better, protocol support. Due to this we have seen in the past years that when looking at request level, rather than just for the initial page, usage is much higher, and this is again the case this year.

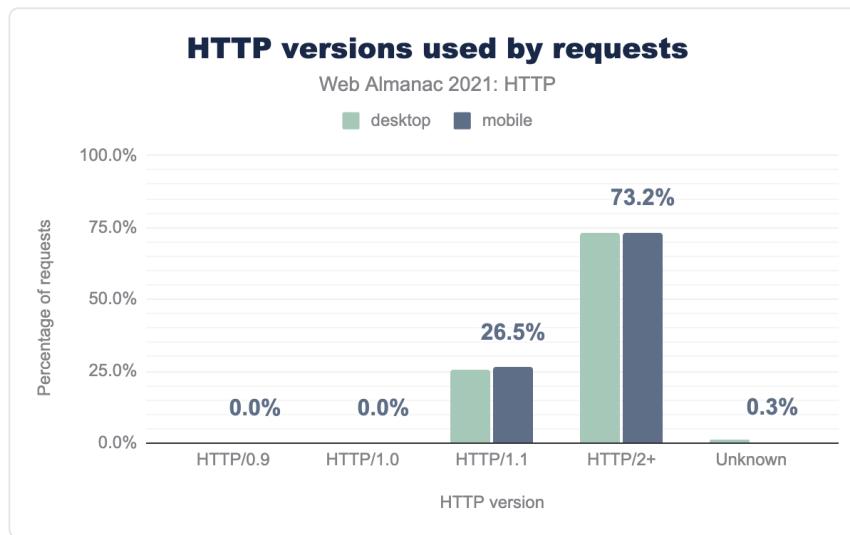


Figure 24.2. HTTP versions used by requests.

In 2021, the HTTP Archive data suggests that HTTP/0.9 and HTTP/1.0 are all but virtually dead. While 0.9 did have hundreds of requests present, this becomes rounded down to zero when aggregated across the entire dataset. HTTP/1.0 has thousands of requests, but it too only represents 0.02% of the total amount.

**25%**

Figure 24.3. Decline in HTTP/1.1 requests in last year.

Interestingly, over a quarter of requests are still served via HTTP/1.1. When compared with 2020, this represents a 25% decline, as 2020 had 50% of requests still leveraging 1.1 across both mobile and desktop. Over 70% of requests are served over HTTP/2 or above, which

suggests that HTTP/2 and HTTP/3 are well and truly the dominant protocol versions for the web.

Looking at the protocol used by page, we can again plot the dominance of HTTP/2 and above:

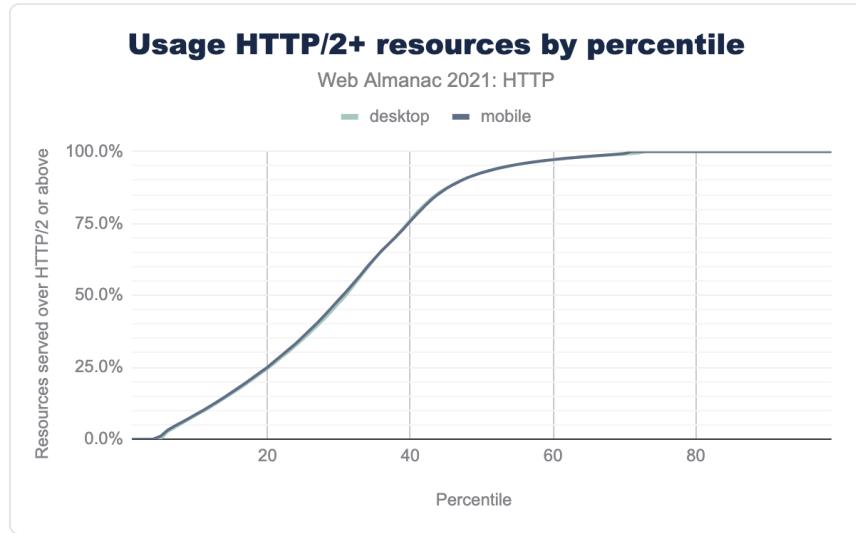


Figure 24.4. Usage HTTP/2+ resources by percentile.

Beyond the 50th percentile of pages, pages have 92% or more of their resources being served over HTTP/2+. And for beyond the 70th percentile 100% of sites resources are loaded over HTTP/2 or better. Put another way, 30% of sites use no HTTP/1.1 resources at all.

## Adoption by third parties

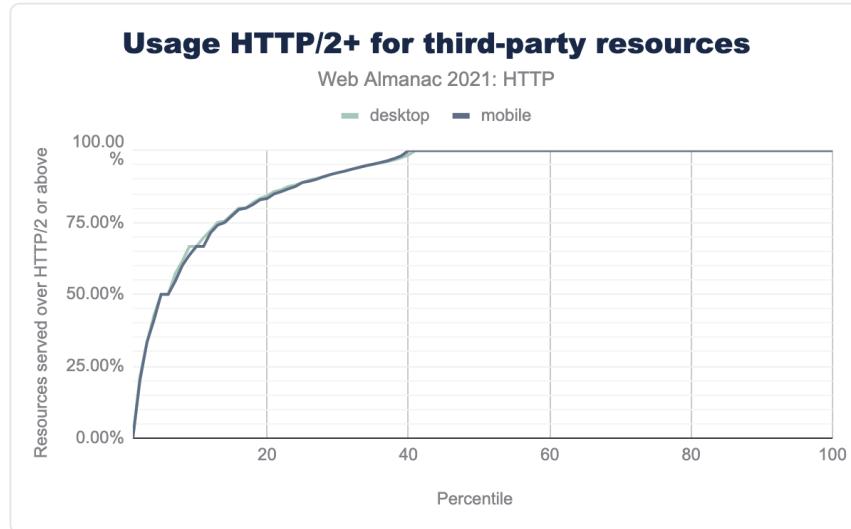


Figure 24.5. Usage HTTP/2+ for third-party resources.

HTTP/2 adoption by third-party content is so heavily skewed, that beyond the 40th percentile of third-party requests, 100% of traffic is being served by HTTP/2. In fact, even at the tenth percentile, over 66% of requests are leveraging HTTP/2. This suggests the majority of adoption is still being influenced by third-party content, and content being served by domains leveraging a CDN.

## Adoption by servers

According to caniuse.com<sup>971</sup>, 97% of browsers support HTTP/2 globally. HTTPS is required by browsers for HTTP/2 support, which may have been a blocker in the past. However, 93% of sites on desktop and 91% on mobile<sup>972</sup> all support HTTPS. This is up 5% from last year in 2020 and was up 6% in the year prior between 2019 and 2020. Implementation of HTTPS is no longer a blocker.

It's important to understand that with such a high adoption across browsers, and high HTTPS adoption, the limiting factor in even greater adoption of HTTP/2 is still largely dictated by the server implementation. Despite the rapid increase in HTTP/2 usage, when you split it out by web server, the adoption figures show a much more fragmented story.

971. <https://caniuse.com/http2>

972. <https://httparchive.org/reports/state-of-the-web#pctHttps>

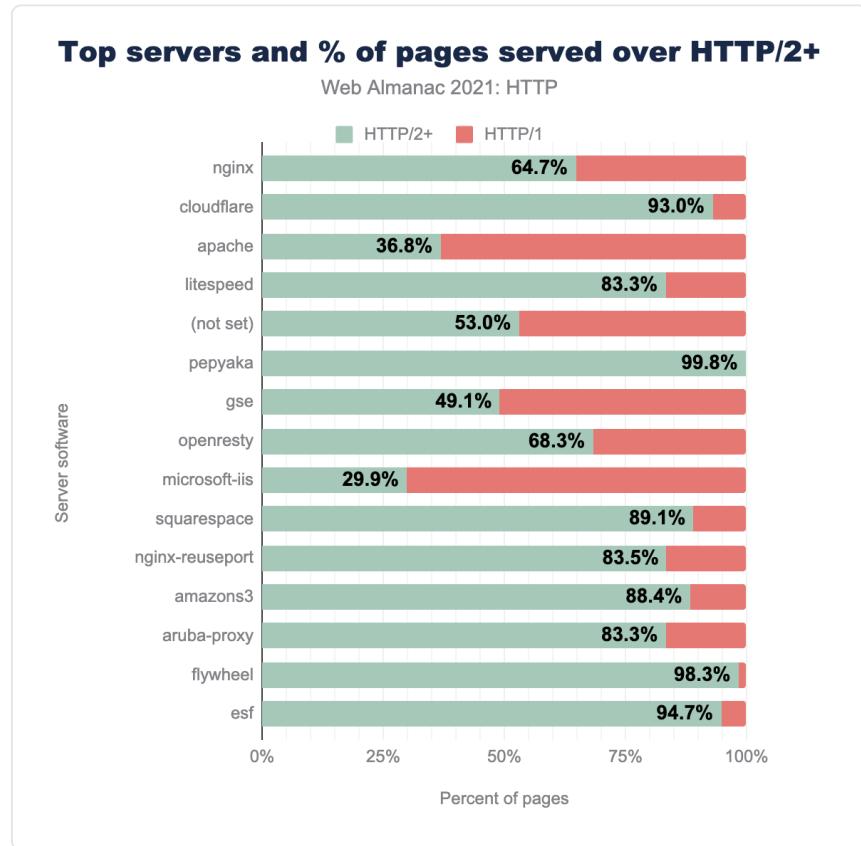


Figure 24.6. Top servers and % of pages served over HTTP/2+.

If a site uses the Apache HTTP server, it is unlikely to have upgraded to HTTP/2, with only one third of Apache servers leveraging the newer protocol. Nginx shows a more promising number with two-thirds of all servers having upgraded to HTTP/2. CDN and cloud servers all promote high adoption rates, from services such as Cloudfront, Cloudflare, Netlify, S3, Flywheel and Vercel. Other niche server implementations such as Caddy or Istio-Envoy also promote good adoption. On the other end of the spectrum, implementations such as IIS, Gunicorn, Passenger, Lighttpd, and Apache Traffic Server (ATS) all have low adoption rates, with Suri also reporting almost zero adoption.

## Server software used by sites not using HTTP/2+

Web Almanac 2021: HTTP

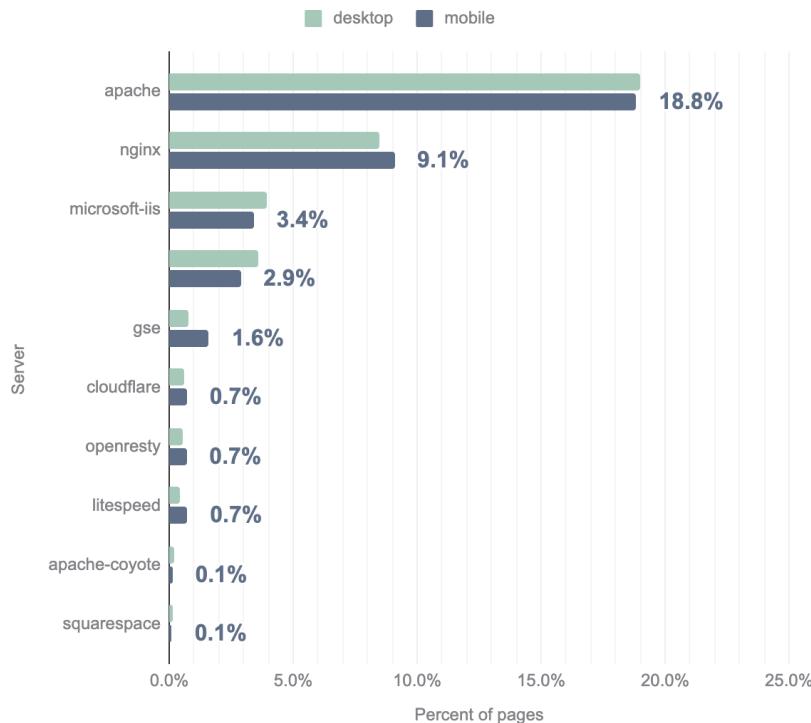


Figure 24.7. Server software used by sites not using HTTP/2+.

In fact, of all servers reporting a HTTP/1.1 response, the server with the largest majority are Apache servers at 20%. As Apache is one of the most popular web servers on the web, it suggests that older installations of Apache may be holding up the web's ability to move forward and adopt the new protocol in full.

### Adoption by CDNs

CDNs are often pivotal to drive adoption of new protocols like HTTP/2, and looking at the stats proves this.



Figure 24.8. Top CDNs and % of pages served over HTTP/2+.

The vast majority of CDNs have 70% or greater adoption of sites with HTTP/2 - much higher than the 49.1% of non-CDN traffic. Some CDNs such as Yottaa, WP Compress and jsDeliver all have 100% adoption of HTTP/2!

The high adopters are typically services around ad networks, analytics, content providers, tag managers, and social media services. The higher adoption of HTTP/2 in these services is clear as even at the fifth percentile and above in which at least 50% of them have enabled HTTP/2. At the median, 95% of these services will be using HTTP/2.

## Adoption by rank

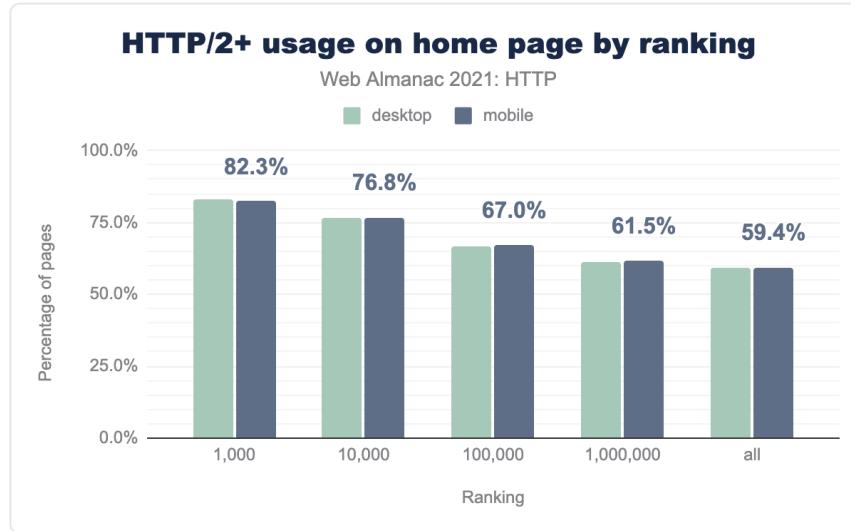


Figure 24.9. HTTP/2+ usage on home page by ranking.

There is also a direct correlation between a site's page rank in the HTTP Archive and its support for HTTP/2. 82% of sites listed in the top 1,000 have HTTP/2 enabled. Over 76% in the top 10k websites, followed by 66% of sites in the top 100k, and at least 60% of sites in the top 1 million will have HTTP/2 enabled. This suggests that higher ranking sites have enabled HTTP/2 for the security and performance benefits offered. The higher ranking a site, the more likely it is to have HTTP/2 enabled.

## Digging a little deeper into HTTP/2

One of main benefits of HTTP/2 is that it is binary instead of a text-based protocol. A request sent over a stream may be made up of one or more *frames*. This changes the mechanics between client and server.

By chunking messages into frames, and interleaving those frames on the wire, a single TCP connection can be used to send and receive multiple messages in one connection. This helps eliminate the need for domain hacks and other HTTP/1.1 performance workarounds.

However, this completely new way of sending HTTP traffic means that HTTP/2 is not compatible with previous versions, and so clients and servers must each know they are talking HTTP/2. HTTPS has been adopted as the de facto standard in HTTP/2. While HTTP/2 can be

implemented without HTTPS, all major browser vendors ensure HTTP/2 is used over HTTPS. HTTP/2 also uses ALPN<sup>973</sup>, which allows for faster-encrypted connections as the protocol can be determined during the initial connection.

## Switching between protocols

While the use of HTTPS can be used to help decide whether to “speak” HTTP/1.1 or the newer HTTP/2, there are other methods of switching to the newer protocol. HTTP/2 support can be advertised on a HTTP/1.1 connection via the `Upgrade` HTTP header, and then the client can use the 101 (Switching Protocols) response status code to make the switch. For HTTP/2 to HTTP/3, a similar `alt-svc` (Alternative Service) header is used, which we will discuss later in this chapter.

The HTTP Archive data suggests that the use of the `Upgrade` header is often misused or configured incorrectly. This feature will in fact be dropped<sup>974</sup> from the next version of HTTP/2. Only a fraction of sites offer the `Upgrade` header at all. The most common header reported is the `h2`, `h2c` detailing the HTTP/2 option, or HTTP/2 over cleartext, with 0.09% of desktop and 0.16% of mobile sites reporting this header.

A similar rate of sites also offer `websockets` as an `Upgrade` option, with 0.08%. Some sites also offer HTTP/1.1 as an upgrade option incorrectly, as `Upgrade` should be used to signal an incompatible or more appropriate protocol other than the existing HTTP/1.1 connection the request was made on. 0.04% of sites also incorrectly report H2 as an `Upgrade` option, despite having this connection already on HTTP/2.

973. [https://en.wikipedia.org/wiki/Application-Layer\\_Protocol\\_Negotiation](https://en.wikipedia.org/wiki/Application-Layer_Protocol_Negotiation)

974. <https://github.com/httpwg/http2-spec/issues/772>



Figure 24.10. Upgrade headers sent over HTTP/2 connections.

More worrying is the number of sites which offer to “upgrade” a HTTP/2 connection to HTTP/2. This is a clear error and used to confuse browsers in the early days of HTTP/2.

There were also almost 120,000 mobile sites found on HTTP, while still reporting an `Upgrade` header to HTTP/2. A better practice would be to issue a redirect from HTTP to HTTPS, and leverage HTTP/2 on the secure connection directly.

# 26,000

Figure 24.11. Mobile websites claiming to support HTTP/2 when they do not.

22,000 and 26,000 web pages on desktop and mobile respectively were also found to be on HTTPS but not support HTTP/2. Similarly, hundreds of web pages were incorrectly signaling to upgrade to HTTP/2 despite the connection already on HTTP/2 itself.

## Number of connections

Since the introduction of HTTP/2 the median number of TCP connections per page has steadily been decreasing.

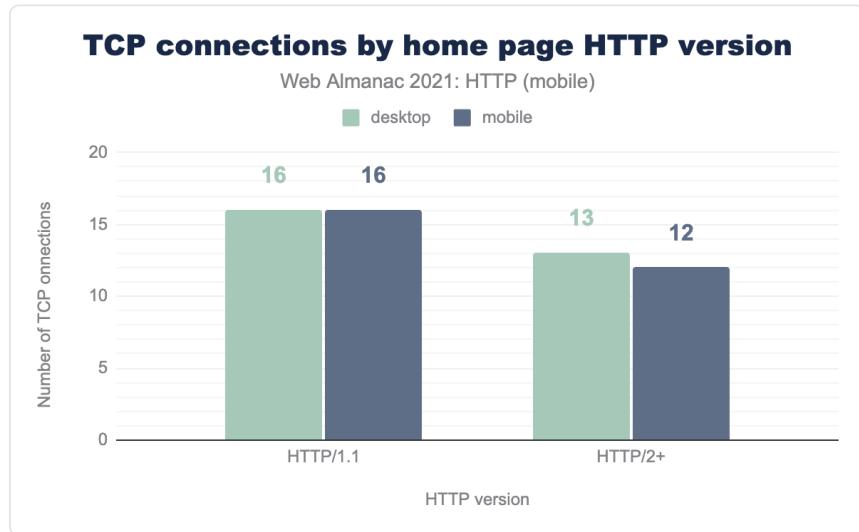


Figure 24.12. TCP connections by home page HTTP version.

At the time of this writing, desktop connections are down 44% over 12 months to a median value of 16 connections. Mobile is down 7% with a median connection count of 12. This represents a good reduction of connections over time, as the adoption of HTTP/2 has increased sharply since 2020.

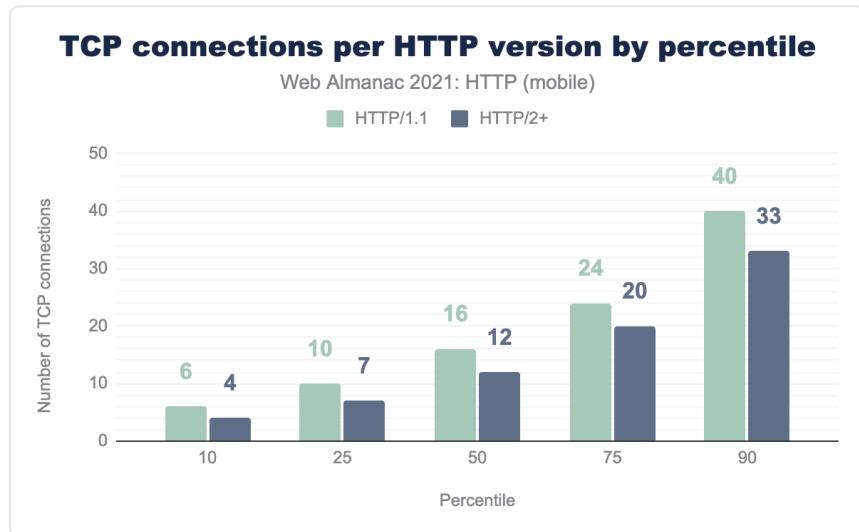


Figure 24.13. TCP connections per HTTP version by percentile.

Based on the HTTP Archive data collected, a median HTTP/1.1 site will have 16 connections per page. Then 24 connections at the 75th percentile. This more than doubles to 40 at the 90th percentile for mobile and desktop. By comparison a HTTP/2 site will have 12 connections on median, 21 connections at 75th percentile, and hits 33 connections at the 90th percentile. Even at the top end, this represents a 21% reduction in the number of connections used across websites.

TLS adds a slight overhead to performance, and with the de facto implementation of HTTP/2 over HTTPS, which means there are performance considerations with the versions of TLS used. Since the introduction of TLS 1.3<sup>975</sup>, extra performance considerations have been added, including TLS false starts<sup>976</sup>, which allows the client to start sending encrypted data immediately after the first TLS round trip. As well as zero round trip time (0-RTT)<sup>977</sup> to improve the TLS handshake. TLS 1.2 needs two round trips to complete TLS handshake, while 1.3 requires only one, which reduces the encryption latency by half.



Figure 24.14. TLS version used by page HTTP version.

The HTTP Archive data suggests that 34% of desktop pages are using TLS 1.2, while 56% are using TLS 1.3, with the remaining 10% unknown (HTTPS sites that failed to connect or similar). This is slightly lower on mobile, with 36% using TLS 1.2, 55% using TLS 1.3 and 9% unknown. While the majority of sites use TLS 1.3, a third of sites on the web could leverage an upgrade to receive these performance boosts.

975. <https://blogs.windows.com/msedgedev/2016/06/15/building-a-faster-and-more-secure-web-with-tcp-fast-open-tls-false-start-and-tls-1-3/>

976. <https://blogs.windows.com/msedgedev/2016/06/15/building-a-faster-and-more-secure-web-with-tcp-fast-open-tls-false-start-and-tls-1-3/>

## Reduce headers

Another feature put forward in HTTP/2 was header compression. HTTP/1.1 proved that there were many duplicate or repeating HTTP headers being sent over the wire. These headers can be particularly large when dealing with cookies. To reduce this overhead, HTTP/2 leverages the HPACK compression format<sup>978</sup> to reduce the size of headers sent and received. Both client and server maintain an index of often used and previously transferred headers in a lookup table and can refer to the index of those values in the table, rather than sending the individual values back and forth. This saves in the number of bytes sent over the wire.

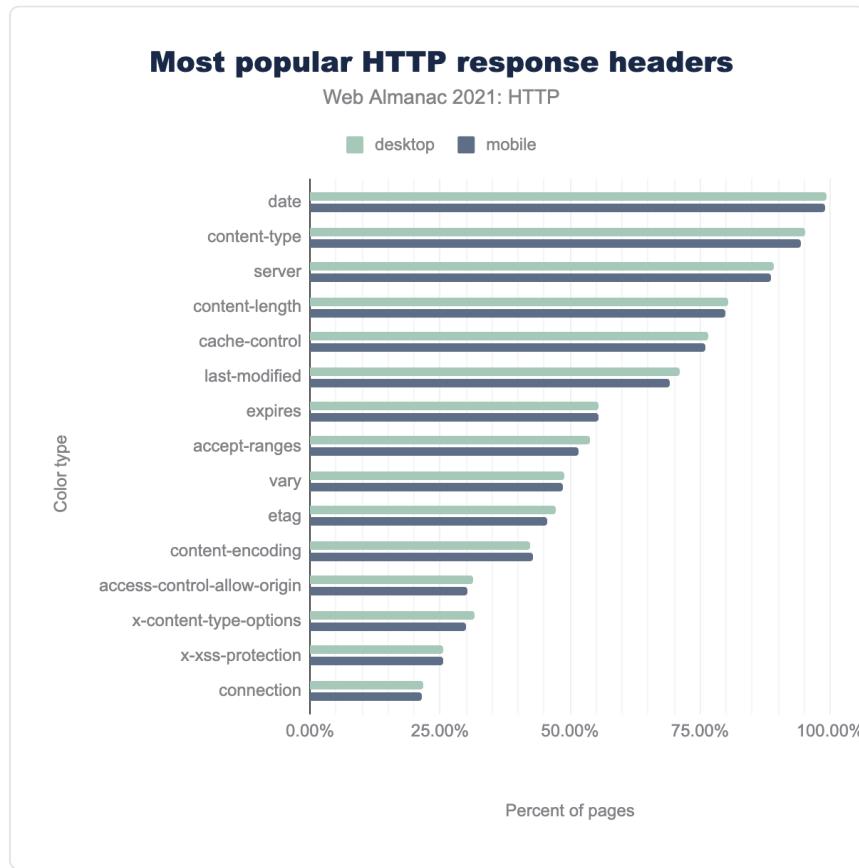


Figure 24.15. Most popular HTTP response headers.

In terms of the most common response headers received, the top five most common headers

978. <https://datatracker.ietf.org/doc/html/rfc7541>

are: `date`, `content-type`, `server`, `cache-control` and `content-length` respectively. The most common non-standard header is Cloudflare's `cf-ray`, followed by Amazon's `x-amz-cf-pop` and `X-amz-cf-id`. Outside of content information (`length`, `type`, `encoding`), caching policies (`expires`, `etag`, `last-modified`) and origin policies (STS, CORS<sup>799</sup>), `expect-ct` reporting certificate transparency and the CSP `report-to` headers are some of the most commonly used headers.

While some of these headers (e.g., `date` or `content-length`) may change with every request, the vast majority will send the same, or a limited number of variations for every request and this is where HTTP/2 header compression can provide benefit. Similarly request headers often send the same data (such as the long `user-agent` header) over and over for every request. Therefore, to consider the impact we must look at the number of requests pages are making.

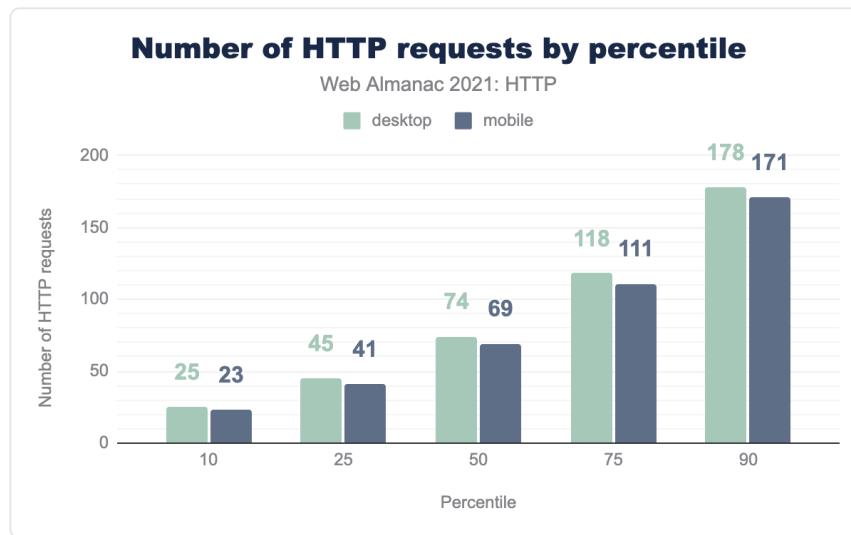


Figure 24.16. Number of HTTP requests by percentile.

The median desktop site has 74 requests, and the median mobile site has 69 requests. Hundreds of sites had over thousands of requests per page. The highest in fact reporting 17,923 requests in total, followed by 10,224. By compressing and reusing the headers sent on previous requests HTTP/2 reduces the impact of repeated requests.

Why our analysis is currently unable to measure the exact impact of Header compression as those details are buried deep in the browser network stack, we can look at the uncompressed header sizes to give some indication of the potential benefit.

799. <https://developer.mozilla.org/docs/Web/HTTP/Headers/Access-Control-Allow-Origin>

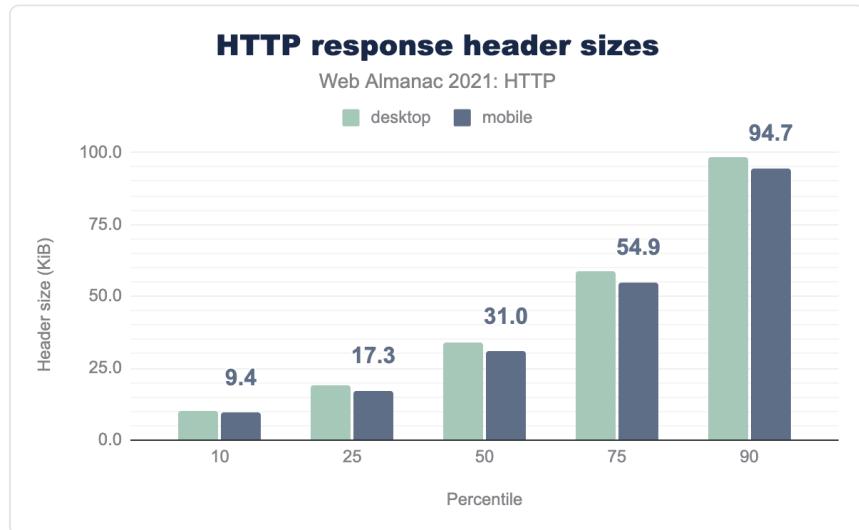


Figure 24.17. HTTP response header sizes.

The median web page returns 34 KB worth of headers for desktop and 31 KB for mobile. At the 90th percentile this increases to 98 KB and 94 KB for desktop and mobile respectively. However, the largest instance of response header was over 5.38 MB. Many sites were discovered having over 1 MB in response headers. Typically, these large response headers are due to overweight `CSP` or `P3P` headers, suggesting the complexities or mismanagement of these headers across websites. In other extreme examples, overweight headers were due to misconfigurations or errors in the application that duplicate multiple `Set-Cookies` or `Cache-Control` settings.

## Prioritization

Streams can also be linked by having one stream depend on another, and they can be weighted by being assigned an integer between 1 and 256. Through these dependencies and weighting scores, the server can prioritize certain key streams, sending their response data before that of other streams.

Since the introduction of HTTP/2, prioritization has been implemented inconsistently across different parts of the web. Andy Davis<sup>980</sup> has found that this inconsistency may create sub-optimal experiences for users on the web. Often this is because servers will ignore prioritizations and serve based on a first-come first-served behavior. In fact, Andy's research

<sup>980</sup>. <https://twitter.com/AndyDavies>

highlights<sup>981</sup> that many of the major CDNs do not implement HTTP/2 prioritization correctly. This also includes a number of the popular cloud load balancers. The 2021 data suggests similar findings as previous years, with only 6 CDNs implementing prioritization correctly. This includes Akamai, Fastly, Cloudflare, Automattic, section.io and Facebook's own CDN.

Patrick Meehan<sup>982</sup> suggests that outside using one of the CDNs that implement prioritization correctly, there are a number of TCP optimizations<sup>983</sup>, including BBR and `tcp_notsent_lowat`, that can be enabled to improve prioritization on the server side.

This inconsistency also exists at the client level, with different browser vendors implementing this behavior differently. Safari implements a static approach to prioritization depending on the asset type and does not map dependencies. Chrome, Edge, and Firefox have a more advanced approach to building out logical dependencies across streams and can reprioritize requested assets on the stream based on the discovered prioritization.

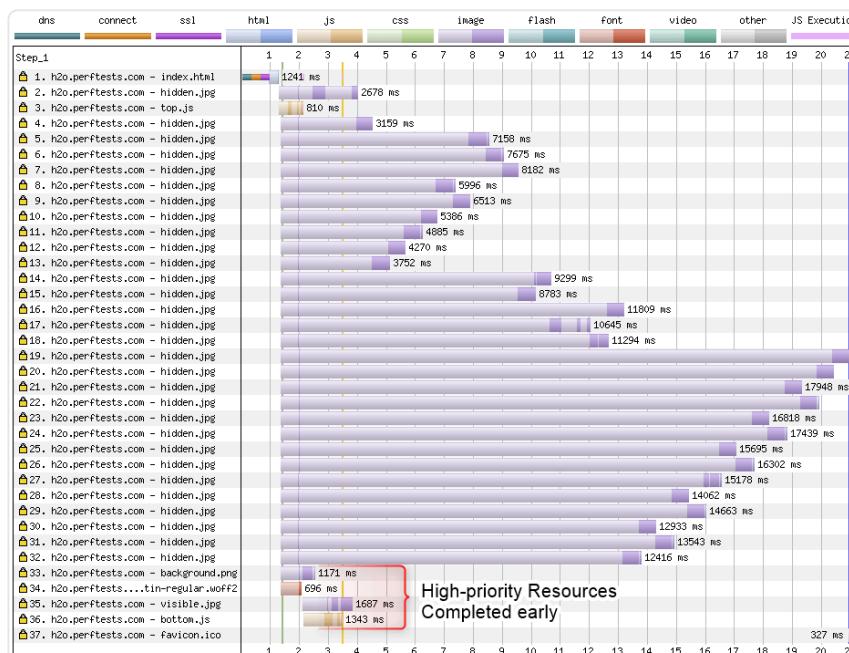


Figure 24.18. WebPageTest waterfall example.

Since HTTP/2 there has been an updated proposal to prioritizations, with the Extensible Prioritization Scheme for HTTP<sup>984</sup> proposal. This includes adding a `priority` header in the

981. <https://github.com/andydavies/http2-prioritization-issues>

982. <https://twitter.com/patmeehan>

983. <https://blog.cloudflare.com/http-2-prioritization-with-nginx/>

984. <https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-priority>

response, as well as a new `PRIORITY_UPDATE` frame for HTTP/2. This `PRIORITY_UPDATE` frame is also proposed for HTTP/3. This has yet to be adopted across the web in full, but has received focus from Cloudflare<sup>985</sup> in an effort to improve the underlying behavior of prioritization<sup>986</sup>.

## The death of HTTP/2 Push?

Another major feature was the introduction of the server push mechanism. HTTP/2 server push allows the server to send multiple resources in response to a client request. Thus, the server informs the client about assets it may need before the client becomes aware they exist. The common use case is to push critical assets such as JavaScript and CSS to the client before the browser has parsed the base HTML and identified those critical assets and subsequently requested them itself. The client also has the option to decline the push message.

Despite the promises of zero round trips, pre-emptive critical assets and the potential for performance upsides, HTTP/2 push has not lived up to the hype.



*Figure 24.19. Sites using HTTP/2 push.*

When analyzed in 2019 HTTP/2 had little adoption, averaging around 0.5%. The following year in 2020, there was an increase to 0.85% adoption across desktop and 1.06% adoption on mobile. This year in 2021 the numbers have slightly increased at 1.03% on desktop, and 1.25% on mobile. Relatively, mobile has seen a significant increase year on year, however at 1.25% overall adoption of HTTP/2 it is still negligible. At the page level, this sits at 64k and 93k requests for desktop and mobile respectively.

<sup>985.</sup> <https://blog.cloudflare.com/better-http-2-prioritization-for-a-faster-web/>  
<sup>986.</sup> <https://blog.cloudflare.com/adopting-a-new-approach-to-http-prioritization/>

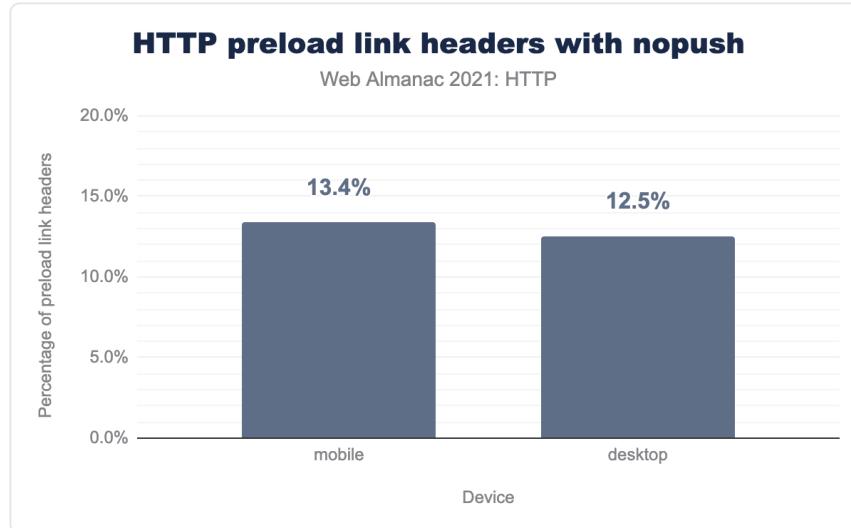


Figure 24.20. HTTP preload link headers with `nopush`.

Many HTTP/2 implementations reused the `preload` resource hint as a signal to push. However, in some cases, a developer may want to preload an asset, but decide they do not want to have it delivered via a HTTP/2 push mechanism. They may want to signal to a CDN or other downstream server to not attempt a push, via the `nopush` directive. This year's data shows that over 200,000 preload headers were used, and on average 12% of those were issued with a `nopush` attribute.

One of the challenges is to implement dynamic push directives at a page level, where the push messages are formed based on the current page and the critical assets for that page, as opposed to a hardcoded series of pushes that apply as a blanket across the site, such as those that may be defined globally in an Nginx<sup>987</sup> or Apache<sup>988</sup> configuration. Despite implementation examples from Akamai<sup>989</sup> and Google<sup>990</sup> that use real user data and analytics to determine this dynamic push configuration, the data shows implementation across the web has been limited. Akamai<sup>991</sup>'s research suggests that when applied correctly, HTTP/2 push provides a clear benefit to web performance.

However, investments made from other CDN providers and server implementations prove that designing for HTTP/2 push is difficult. In fact Jake Archibald<sup>992</sup> described some of these challenges<sup>993</sup> back in 2017. These focus on problems with push cache, browser inconsistencies,

987. <https://www.nginx.com/blog/nginx-1-13-9-http2-server-push/>

988. <https://httpd.apache.org/docs/2.4/howto/http2.html#push>

989. <https://medium.com/@anneric/http-2-server-push-performance-a-further-akamai-case-study-7a17573a3317>

990. <https://github.com/guess-js/guess/>

991. <https://medium.com/@anneric/http-2-server-push-performance-a-further-akamai-case-study-7a17573a3317>

992. <https://twitter.com/jaffathecake>

993. <https://jakearchibald.com/2017/h2-push-tougher-than-i-thought/>

and superfluous bytes sent from the server if the client determines the push isn't needed.

Attempts to resolve some<sup>994</sup> of these<sup>995</sup> issues were abandoned, largely due to issues around privacy and security concerns, where cache digests may be used to identify users.

Patrick Meehan breaks down some of the problems in this post on a possible alternative - 103 Early Hints<sup>996</sup>. In that post he details that Push usually ends up delaying HTML and other render blocking assets.

## Pushed assets

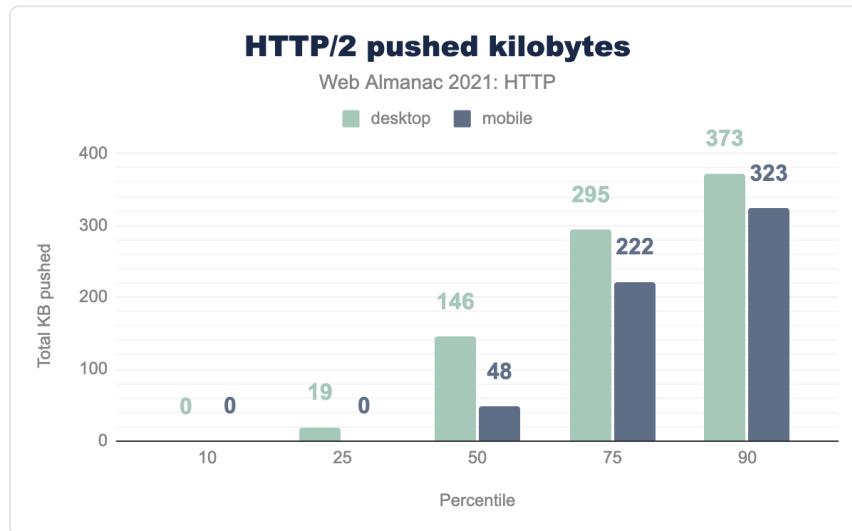


Figure 24.21. HTTP/2 pushed kilobytes.

In cases where items were pushed, the median size of the bytes that were pushed were 145 KB for desktop and 48 KB for mobile. This almost doubles to 294 KB for desktop and more than quadruples for mobile at 221 KB for the 75th percentile. At the top end, we see 372 KB pushed and 323 KB for mobile at the 90th percentile.

While these numbers at the 90th percentile appear fine, it's when you start to review the number of pushes, it highlights the misuse of the push feature:

<sup>994</sup>. <https://datatracker.ietf.org/doc/html/draft-ietf-httpbis-cache-digest#appendix-A>

<sup>995</sup>. <https://datatracker.ietf.org/doc/html/draft-vkrasnov-h2-compression-dictionaries-03>

<sup>996</sup>. <https://blog.cloudflare.com/early-hints/#--text=summarized%20server%20push%E2%80%99s%20gatchas>



Figure 24.22. HTTP/2 pushed kilobytes.

The median number of pushes is 4 and 3 across desktop and mobile respectively. This moves to 8 at the 75% percentile and jumps to 21 and 16 at the 90th percentile. The 100% percentile sees an amazing 517 and 630 pushes being done by some sites, which highlights the dangers of the feature, particularly when considering push was originally designed to advertise a small number of critical assets early in the request.



Figure 24.23. HTTP/2 pushed counts.

When analyzing by content type, the data suggests that fonts are the most commonly pushed asset, followed by images, CSS, scripts and video. These numbers paint a different story when looking at the size of the asset types. Fonts are still the largest assets pushed by volume, but scripts are not far behind. This is followed by images, videos and then CSS. Therefore, this suggests that despite more CSS files being pushed, they are small in size. Scripts aren't pushed as often as fonts, images and CSS, but represent a larger volume of the push data.

As the numbers above suggest, and as described in previous years, HTTP push is underutilized. When utilized, it is often misused or not used in the intended manner, which is likely to be a performance detriment for the end user.

Google has flagged its intent to remove push from Chrome. However, throughout 2021 there was still ongoing debate<sup>997</sup> around the efficacy of HTTP/2 Push. This removal is yet to happen, and it is largely suggested that Push can be leveraged through CDNs who implement it correctly. Google recommends leveraging the `<link rel="preload">` directive as an alternative to push, albeit this still incurs a 1 RTT, which is what push aims to solve. Google also reports<sup>998</sup> it has not implemented Push in HTTP/3, and neither have others such as Cloudflare.

### An alternative to push

The other commonly suggested alternative to Push is the use of *Early Hints*. This works by

<sup>997</sup> <https://groups.google.com/a/chromium.org/g/blink-dev/c/K3rYLvmQUBY/m/vOWBKZGoAQAJ>

<sup>998</sup> <https://groups.google.com/a/chromium.org/g/blink-dev/c/K3rYLvmQUBY/m/vOWBKZGoAQAJ>

having the server report a `103` status code response message, with `preload` hints in the Link header. Early Hints allows the server to report on assets that the client should `preload` before getting the page HTML back.

### HTTP/1.1 103 Early Hints

```
Link: <style.css>; rel="preload"; as="style"
```

CDNs such as Fastly<sup>999</sup> and Cloudflare<sup>1000</sup> have been experimenting with early hints, but it's still early days for Early Hints. At the time of this writing, Early Hints support in HTTP/2 inside Chrome is still being worked on<sup>1001</sup>, and while other browser vendors have announced support for Early Hints, and while Cloudflare has introduced support in the wild, many other vendors have not yet made concrete implementations.

Despite incremental adoption for HTTP/2 push year on year, it is likely that Google and other browser vendors abandon support for push, in favor of alternatives such as Early Hints. Coupled with support from CDNs, Early Hints is likely to be the replacement. Last year, we proposed the question of whether it was a goodbye to HTTP/2 push. This year we suggest that mainstream use of HTTP/2 is dead, at least for the web browsing use case.

## HTTP/3

HTTP/3 is the next advancement of HTTP/2 and builds upon its foundation with even more changes down throughout the protocol. The biggest change is the move away from TCP to a UDP-based transport protocol called QUIC. This allows quicker advancements in HTTP, without waiting for TCP implementations that are ingrained all across the internet to support them. For example, HTTP/2 introduced the concept of independent streams but, at a TCP level these were still part of one TCP stream, and so not truly independent. Changing TCP to support this would take considerable time before it would be so widely supported as to be safe to use. Therefore HTTP/3 switches to an alternative transport protocol. QUIC is similar to TCP in many ways, and basically re-builds all the many useful features of TCP, but with the addition of new features. QUIC is encrypted and delivered over the well-supported, lightweight UDP transport protocol.

999. <https://www.fastly.com/blog/beyond-server-push-experimenting-with-the-103-early-hints-status-code>

1000. <https://blog.cloudflare.com/early-hints/>

1001. <https://bugs.chromium.org/p/chromium/issues/detail?id=671310>

## HTTP/3 Adoption

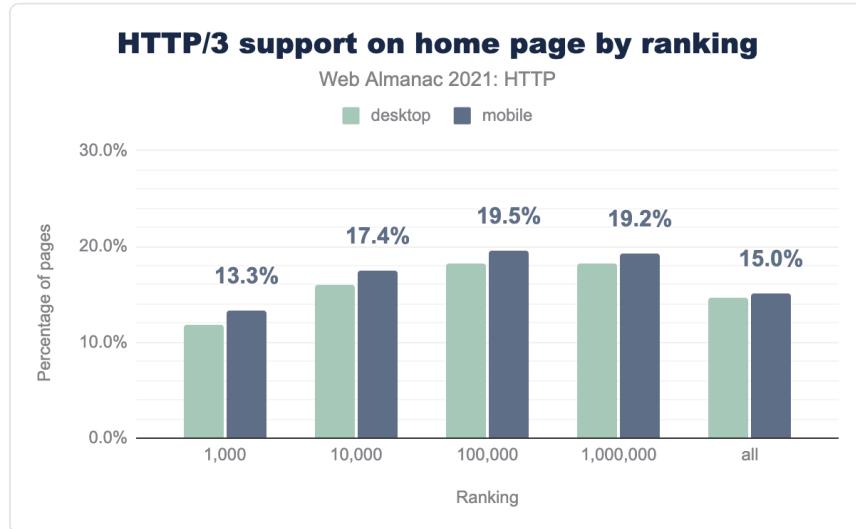


Figure 24.24. HTTP/3 support on home page by ranking.

Earlier in the chapter we found that sites that were ranked higher had greater adoption of HTTP/2. Surprisingly, the opposite is true of HTTP/3. We see less support from the top one thousand sites than we do the top one million, with slightly more support implemented across mobile sites.

Distribution across the top one hundred thousand sites and top one million sites at 18% and 19% for desktop and mobile respectively. This drops to 16% and 17% within the top ten thousand sites. The top one thousand sees 11% and 13% deployment across desktop and mobile. Adoption beyond the top one million sit around 15% for implementation across homepages. Overall, this is quite a strong adoption across the board, likely spearheaded by the support from some of the major CDNs. This suggests that while the top websites have adopted HTTP/2 as mainstream, many have yet to explore HTTP/3.

## HTTP/3 Support

Web server support for HTTP/3 is still limited in the market. Nginx represents the most common HTTP server on the web, with about two thirds of HTTP/2 sites using a version of Nginx. Nginx has publicly expressed support for HTTP/3, including discussing their roadmap<sup>1002</sup> to roll out full support, and aim to have full support by the end of 2021. The Apache server, by

1002. <https://www.nginx.com/blog/our-roadmap-quic-http-3-support-nginx/>

comparison, has yet to provide any guidance on when HTTP/3 will be supported. Microsoft has announced support for HTTP/3 in its new Windows Server 2022<sup>1003</sup>. Other alternatives such as the LiteSpeed web server have leaned into its support<sup>1004</sup> for HTTP/3, whereas Caddy has enabled support for HTTP/3 as an experimental feature<sup>1005</sup> available. Node.js support is held up<sup>1006</sup> due to lack of OpenSSL support.

A number of CDNs have also expressed support for HTTP/3. Cloudflare has been experimenting with HTTP/3 since 2019<sup>1007</sup>, in which they report better performance in many examples. Cloudflare have also published their quiche<sup>1008</sup> library, which powers their HTTP/3 deployment on the edge network. Fastly has also discussed its support<sup>1009</sup> for HTTP/3, and has it available as a BETA service<sup>1010</sup>. Fastly have also open sourced their own implementation known as quickly<sup>1011</sup>, designed for the H2O HTTP<sup>1012</sup> server that Fastly uses on their edge network. Akamai has also expressed continued support<sup>1013</sup> for HTTP/3 and QUIC, and has worked with Microsoft to fork a version of OpenSSL with QUIC<sup>1014</sup> to help move support forward<sup>1015</sup>.

Browser support for HTTP/3 is still evolving. As of October 2021, support is available in the most recent version of Microsoft Edge, Firefox, Google Chrome, and Opera, and partially across mobile for some Android variants and Opera mobile. Support from Safari is limited on macOS 11 Big Sur and must be enabled via the “Experimental Features”, support for iOS is also only available as an experimental feature behind a flag.

## Negotiating HTTP/3

As HTTP/3 is on a completely different transport layer to traditional TCP-based HTTP it is not possible to negotiate HTTP/3 as part of the connection set up—like what happens with HTTP/2 through the HTTPS negotiation. By that stage you have already picked your transport protocol!

HTTP/3 instead requires the `alt-svc` header. You start on a TCP-based HTTP connection (presumably HTTP/2 if the client is advanced enough to support HTTP/3), and then the server can signal through the `alt-svc` header on responses to any requests, that this server also support HTTP/3 over UDP and QUIC. The browser can then decide to try to connect via that. Due to the several iterations of HTTP/3, this header is also how client and server can decide which version of HTTP/3 they decide on.

1003. <https://blog.workinghardinitwork.com/2021/10/11/iis-and-http-3-quic-tls-1-3-in-windows-server-2022/>

1004. <https://docs.litespeedtech.com/cp/cpanel/quic-https/>

1005. <https://caddyserver.com/docs/caddyfile/options>

1006. <https://github.com/nodejs/node/pull/37067>

1007. <https://blog.cloudflare.com/http3-the-past-present-and-future/>

1008. <https://github.com/cloudflare/quiche>

1009. <https://www.fastly.com/blog/why-fastly-loves-quic-http3>

1010. <https://www.fastly.com/blog/modernizing-the-internet-with-http3-and-quic>

1011. <https://github.com/h2o/exemplar>

1012. <https://h2o.example.net/>

1013. <https://www.akamai.com/blog/performance/http3-and-quic-past-present-and-future>

1014. <https://github.com/quiclets/openssl>

1015. <https://daniel.haxx.se/blog/2021/10/25/the-quic-api-openssl-will-not-provide/>

So, in the very first case, HTTP/2 will be used in the initial request, and once the browser discovers the `alt-svc` header, it can then switch protocols and start using HTTP/3. For future cases the browser can cache the `alt-svc` header, and next time jump straight to trying HTTP/3.

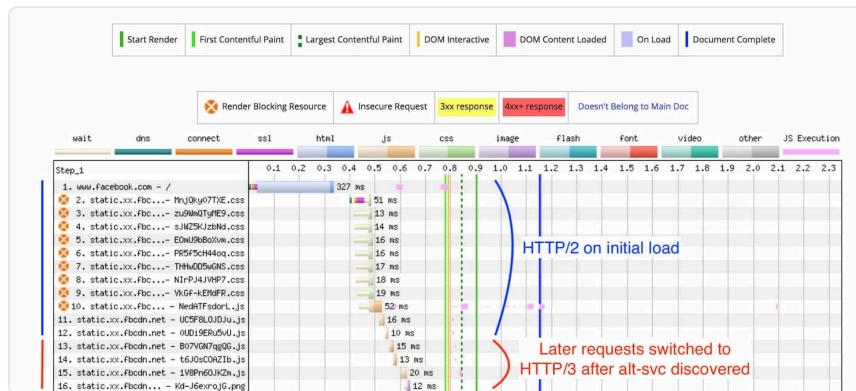


Figure 24.25. WebPageTest example showing HTTP2 switching to HTTP3 during page load.

Also, due to connection coalescing (connection reuse), in some instances if two hostnames resolve over DNS to the same IP and use the same TLS certificate and version, then the client could reuse the same connection across both hostnames. Therefore, it is not uncommon to see a waterfall request with a mix of both HTTP/2 and HTTP/3, depending on the number of hosts and TLS certificates used.

At a page level, about 15% of requests offer an `alt-svc` header. These vary between syntax that offer QUIC, one of the various H3 pre-release versions (officially HTTP/3 is not standardized at the time of writing, but it's in the very final stages). Some sites will advertise support for multiple versions of QUIC, for example `quic=":443"; ma=2592000; v="39,43,46,50"`, while some will only offer one version. The most common advertisement of the `alt-svc` is `"h3-27=:443"; ma=86400, h3-28=:443"; ma=86400, h3-29=:443"; ma=86400, h3=:443"; ma=86400"`, across 11% of all `alt-svc` responses. This header instructs clients that it supports HTTP/33 versions 27, 28 and 29, with a `max-age` of 24 hours.

In instances where `alt-svc` was present, most sites were appending version numbers as they adopt support for new protocol versions, however there were many cases where sites were using the `clear` directive to invalidate previously advertised support.

At the time of this writing the most recent version of the HTTP/3 spec<sup>1016</sup> is version 34. However,

1016. <https://datatracker.ietf.org/doc/html/draft-ietf-quic-http-34>

only 0.01% of responses report this latest version. When viewing details of `alt-svc` at a request level, version 27 is the most commonly requested version in response headers. The server will indicate the preferred versions in order from left to right. 6% of requests will report `h3-27` in the first instance preferred, with 28 and 29 as alternate versions offered in the same response. 2% of responses will offer `h3-29` as the only preferred version for upgrade. QUIC as the preferred protocol update, receives a mere 0.11%, mostly due to outdated servers reporting this incorrectly. In reality there were little differences technically from `h3-29` onwards and most implementations froze versions at that, awaiting the official launch of `h3`.

Most `alt-svc` reported a `max-age` of only 24 hours, which is the default if not specified. The longest `max-age` reported for `alt-svc` was 30 days or 2592000 seconds.



Figure 24.26. WebPageTest `alt-svc` example.

## HTTP/3 considerations and concerns

While many of the upsides of HTTP/3 have been discussed, there are also some concerns and criticisms that have been raised. Many developers are only now comfortable with the changes introduced from HTTP/2, after having to roll back many web performance workarounds to overcome the limitations from HTTP/1.1, as those workarounds later became anti-patterns<sup>2017</sup> in HTTP/2.

In some cases, developers and site owners may argue that the incremental gains from HTTP/3 may not be worth major upgrades to their web servers. Particularly when HTTP/3 hasn't solved all the problems identified in HTTP/2, such as prioritization or effective use of server push. As

2017. <https://docs.google.com/presentation/d/1r7QXGYOLCh4fcUq0jDdDwKJWNqWK1o4xMtYpKZCJyJM/present?slide=id.p19>

such, adoption may be driven at the CDN level, and not within web applications. This may particularly be the case if some servers may not support HTTP/3 or be blocked by lack of OpenSSL support.

As discussed throughout this chapter, QUIC relies on the UDP protocol. With the introduction of HTTP/3, UDP traffic is due to increase across the web. However, currently UDP is often used as an attack vector, such as those in a reflection attack<sup>1018</sup>. QUIC does have some protection mechanisms<sup>1019</sup> in place, but this may mean changes to the way UDP is treated across the web, and the amount of UDP traffic allowed on some networks and firewalls. In the same instance, there may be adoption pushback in cases where TCP headers and the unencrypted parts of the packet are used by firewalls and other middleboxes<sup>1020</sup> across the web. As QUIC encrypts more parts of the packet, there is less visibility for inspection on the packet, and may limit how these middleboxes operate, including the ability to do additional security checks.

There are also concerns that QUIC may be a performance problem on the server side. This is because of higher CPU requirements needed when dealing with UDP. Some estimates suggest twice as much CPU is needed when compared with HTTP/2. This said, there are a number of attempts to optimize QUIC CPU performance<sup>1021</sup> ongoing.

Despite these concerns, the real benefits will be received from the web's end users. QUIC's ability to maintain connections, when switching network connections, allowing for a mobile-first experience in a mobile-first world. The improvements to head-of-line blocking will also ensure greater gains in page load, where we all now know that every millisecond<sup>1022</sup> counts. The enhanced encryption QUIC introduces also allows for a more safe and secure web. As well as the 0-RTT possible with HTTP/3 allows for improved performance.

## Conclusion

Throughout this chapter we have looked at the evolution of HTTP, with a primary focus on the increasing adoption of HTTP/2, and the benefits the newer protocol version offers. This was followed by a closer look at HTTP/3 and how version 3 aims to solve many of the concerns identified after several years of HTTP/2 use across the web.

The HTTP Archive data suggests that this year saw a major uptake in adoption of HTTP/2, with 72% of requests using HTTP/2, and 59% of base HTML pages using HTTP/2. This adoption is largely fueled by increased adoption from CDN providers. HTTP/1.1 is now in the minority across the web.

<sup>1018</sup> <https://blog.cloudflare.com/reflections-on-reflections/>

<sup>1019</sup> <https://datatracker.ietf.org/doc/html/draft-ietf-quic-transport-27#section-8.1>

<sup>1020</sup> <https://en.wikipedia.org/wiki/Middlebox>

<sup>1021</sup> [https://conferences.sigcomm.org/sigcomm/2020/files/slides/epia/0%20QUIC%20and%20HTTP\\_3%20CPU%20Performance.pdf](https://conferences.sigcomm.org/sigcomm/2020/files/slides/epia/0%20QUIC%20and%20HTTP_3%20CPU%20Performance.pdf)

<sup>1022</sup> <https://ai.googleblog.com/2009/06/speed-matters.html>

Despite the uptake on HTTP/2, the push features of HTTP/2 remain underutilized, due to the complexities of implementation, and we suggest that push may be in fact dead on arrival. At the same time, we have seen ongoing concerns with resource prioritization, and incorrect implementations outside the major CDN vendors. Complexities with prioritization remain so prevalent that it has been removed from the HTTP/3 specification.

2021 also allowed us to take a closer inspection on the adoption of HTTP/3. Major players such as Google and Facebook have been rolling out their own support for HTTP/3 for a number of years. Wider adoption of HTTP/3 has been influenced by Akamai, Cloudflare, and Fastly who have publicly been working to support HTTP/3 for other parts of the web.

HTTP/3 aims to build upon the improvements of HTTP/2, including the head-of-line blocking imposed by TCP, while also ensuring more parts of the protocol stack are secure with QUIC's tighter encapsulation of TLS 1.3. However, it is still early days for HTTP/3. We look forward to measuring the adoption of HTTP/3 in 2022, and believe it is likely to gain further traction as support for HTTP/2 becomes mainstream and people look to gain further improvements over current deployments.

There are some concerns expressed with HTTP/3, but any of these concerns should be outweighed by performance gained by the end user. It is likely the HTTP/3 adoption will also be fueled by CDN rollouts, as they work towards their own implementations, as we saw with HTTP/2. Particularly we are yet to see implementations across major web frameworks. It is also likely that we will see a mix of HTTP/2 and HTTP/3 over the next several years.

---

## Author

---



Dominic Lovell

 @dominiclovell  dominiclovell

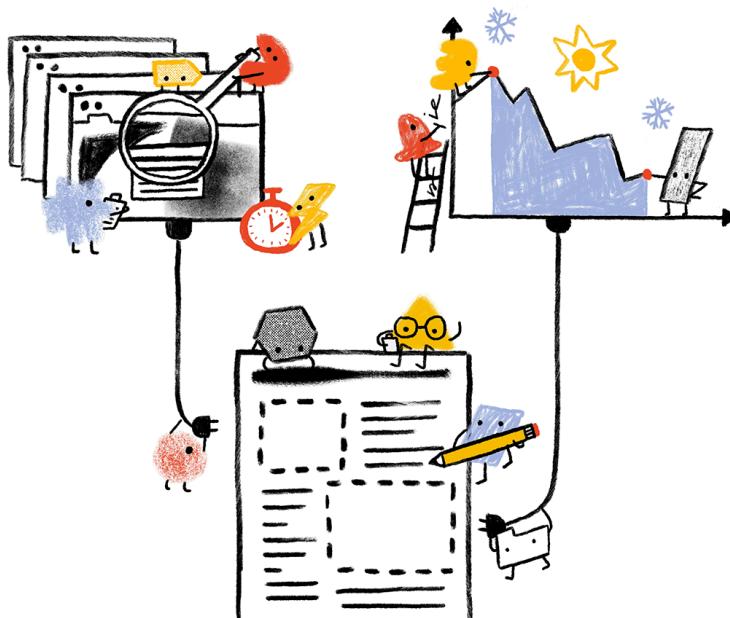
Dominic Lovell is currently a Solutions Engineering Manager at Akamai Technologies, and has been working for a number of years to make sites more performant and safer across the web. You can find him tweeting @dominiclovell, or you can connect with him on LinkedIn<sup>1023</sup>.

---

1023. <https://www.linkedin.com/in/dominiclovell/>

# Appendix A

# Methodology



## Overview

The Web Almanac is a project organized by HTTP Archive<sup>1024</sup>. 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 BigQuery<sup>1025</sup>.

The Web Almanac's mission is to become an annual repository of public knowledge about the state of the web. Our goal is to make the data warehouse of HTTP Archive even more

1024. <https://httparchive.org>

1025. <https://httparchive.org/faq#how-do-i-use-bigquery-to-write-custom-queries-over-the-data>

accessible to the web community by having subject matter experts provide contextualized insights.

The 2021 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 2021 edition of the Web Almanac, unless otherwise noted in the chapter, all metrics were sourced from the July 2021 crawl. These results are publicly queryable<sup>1026</sup> on BigQuery in tables prefixed with `2021_07_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 repository<sup>1027</sup>.

*Please note that some of these queries are quite large and can be expensive<sup>1028</sup> to run yourself. For help controlling your spending, refer to Tim Kadlec's post Using BigQuery Without Breaking the Bank<sup>1029</sup>.*

For example, to understand the median number of bytes of JavaScript per desktop and mobile page, see `bytes_2021.sql`<sup>1030</sup>:

```
#standardSQL
# Sum of JS request bytes per page (2021)
SELECT
    percentile,
    _TABLE_SUFFIX AS client,
    APPROX_QUANTILES(bytesJs / 1024, 1000)[OFFSET(percentile * 10)] AS js_kilobytes
FROM
```

1026. [https://github.com/HTTPArchive/httparchive.org/blob/main/docs/gettingstarted\\_bigquery.md](https://github.com/HTTPArchive/httparchive.org/blob/main/docs/gettingstarted_bigquery.md)  
 1027. <https://github.com/HTTPArchive/almanac.httparchive.org/tree/main/sql/2021>

1028. <https://cloud.google.com/bigquery/pricing>  
 1029. <https://timkadlec.com/remembers/2019-12-10-using-bigquery-without-breaking-the-bank/>  
 1030. [https://github.com/HTTPArchive/almanac.httparchive.org/blob/main/sql/2021/javascript/bytes\\_2021.sql](https://github.com/HTTPArchive/almanac.httparchive.org/blob/main/sql/2021/javascript/bytes_2021.sql)

```

`httparchive.summary_pages.2021_07_01_*` ,
UNNEST([10, 25, 50, 75, 90, 100]) AS percentile
GROUP BY
percentile,
client
ORDER BY
percentile,
client

```

Results for each metric are publicly viewable in chapter-specific spreadsheets, for example JavaScript results<sup>1031</sup>. Links to the raw results and queries are available at the bottom of each chapter. Metric-specific results and queries are also linked directly from each figure.

## Websites

There are 8,198,531 websites in the dataset. This represents an increase of 9% compared to the 2020 edition of the Web Almanac. Among those, 7,499,763 are mobile websites and 6,294,605 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 July 2021 HTTP Archive crawl used by the Web Almanac used the most recently available Chrome UX Report release for its list of websites. The 202105 dataset was released on June 8, 2021 and captures websites visited by Chrome users during the month of May.

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.

---

<sup>1031</sup> <https://docs.google.com/spreadsheets/d/1zU9rHpl3nC6jTz3xgN6w13afW7x34xAKBh2IPH-lVxk/edit#gid=18398250>

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

WebPageTest<sup>1032</sup> is a prominent web performance testing tool and the backbone of HTTP Archive. We use a private instance<sup>1033</sup> of WebPageTest with private test agents, which are the actual browsers that test each web page. Desktop and mobile websites are tested under

---

1032. <https://www.webpagetest.org/>  
1033. <https://docs.webpagetest.org/private-instances/>

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/91.0.4472.114 Safari/537.36 PTST/210702.163639	Mozilla/5.0 (Linux; Android 6.0.1; Moto G (4)) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/91.0.4472.114 Mobile Safari/537.36 PTST/210702.163639
Location	Google Cloud Locations, USA	Google Cloud Locations, USA
Connection	Cable (5/1 Mbps 28ms RTT)	3G (1.600/0.768 Mbps 300ms RTT)
Viewport	1376 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.

Test agents run from various Google Cloud Platform locations<sup>1034</sup> based in the USA.

HTTP Archive's private instance of WebPageTest is kept in sync with the latest public version and augmented with custom metrics<sup>1035</sup>, which are snippets of JavaScript that are evaluated on each website at the end of the test.

The results of each test are made available as a HAR file<sup>1036</sup>, a JSON-formatted archive file containing metadata about the web page.

## Lighthouse

Lighthouse<sup>1037</sup> 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

1034. <https://cloud.google.com/compute/docs/regions-zones/#locations>

1035. <https://github.com/HTTPArchive/custom-metrics>

1036. [https://en.wikipedia.org/wiki/HAR\\_file\\_format](https://en.wikipedia.org/wiki/HAR_file_format)

1037. <https://developers.google.com/web/tools/lighthouse/>

pages are not included because of limited resources. As of the July 2021 crawl, HTTP Archive used a combination of 8.0.0<sup>1038</sup> and 8.1.0<sup>1039</sup> versions 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 documentation<sup>1040</sup>.

## Wappalyzer

Wappalyzer<sup>1041</sup> is a tool for detecting technologies used by web pages. There are 90 categories<sup>1042</sup> of technologies tested, ranging from JavaScript frameworks, to CMS platforms, and even cryptocurrency miners. There are over 2,600 supported technologies (an increase from 1,400 last year).

HTTP Archive runs the latest version of Wappalyzer for all web pages. As of July 2021 the Web Almanac used the 6.7.7 version<sup>1043</sup> 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 Ecommerce<sup>1044</sup> and CMS<sup>1045</sup> 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.

1038. <https://github.com/GoogleChrome/lighthouse/releases/tag/v8.0.0>

1039. <https://github.com/GoogleChrome/lighthouse/releases/tag/v8.1.0>

1040. <https://developers.google.com/web/tools/lighthouse/>

1041. <https://www.wappalyzer.com/>

1042. <https://www.wappalyzer.com/technologies>

1043. <https://github.com/AlasO/Wappalyzer/releases/tag/v6.7.7>

1044. <https://www.wappalyzer.com/categories/ecommerce>

1045. <https://www.wappalyzer.com/categories/cms>

## Chrome UX Report

The Chrome UX Report<sup>1046</sup> 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.).

As of this year, the Chrome UX Report dataset now includes relative website ranking data<sup>1047</sup>. These are referred to as *rank magnitudes* because, as opposed to fine-grained ranks like the #1 or #116 most popular websites, websites are grouped into rank buckets from the top 1k, top 10k, up to the top 10M. Each website is ranked according to the number of eligible<sup>1048</sup> page views on all of its pages combined. This year's Web Almanac makes extensive use of this new data as a way to explore variations in the way the web is built by site popularity.

For Web Almanac metrics that reference real-world user experience data from the Chrome UX Report, the July 2021 dataset (202107) is used.

You can learn more about the dataset in the Using the Chrome UX Report on BigQuery<sup>1049</sup> guide on [web.dev](https://web.dev)<sup>1050</sup>.

## Blink Features

Blink Features<sup>1051</sup> are indicators flagged by Chrome whenever a particular web platform feature is detected to be used.

We use Blink Features to get a different perspective on feature adoption. This data is especially useful to distinguish between features that are implemented on a page and features that are actually used. For example, the CSS chapter's section on Grid layout uses Blink Features data to measure whether some part of the actual page layout is built with Grid. By comparison, many more pages happen to include an unused Grid style in their stylesheets. Both stats are interesting in their own way and tell us something about how the web is built.

Blink Features are reported by WebPageTest as part of our regular testing.

1046. <https://developers.google.com/web/tools/chrome-user-experience-report>

1047. <https://developers.google.com/web/updates/2021/03/crx-rank-magnitude>

1048. <https://developer.chrome.com/docs/crx/methodology/#eligibility>

1049. <https://web.dev/chrome-ux-report-bigquery>

1050. <https://web.dev/>

1051. [https://chromium.googlesource.com/chromium/src/+/HEAD/docs/use\\_counter\\_wiki.md](https://chromium.googlesource.com/chromium/src/+/HEAD/docs/use_counter_wiki.md)

## Third Party Web

Third Party Web<sup>1052</sup> 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.

Several chapters in the Web Almanac use the domains and categories from this dataset to understand the impact of third parties.

## Rework CSS

Rework CSS<sup>1053</sup> 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 thread<sup>1054</sup> for more information about how it was integrated with the HTTP Archive dataset on BigQuery.

## Rework Utils

This year's CSS chapter revisits many of the metrics introduced in last year's CSS chapter, which was led by Lea Verou. Lea wrote Rework Utils<sup>1055</sup> to more easily extract insights from Rework CSS's output. Most of the stats you see in the CSS chapter continue to be powered by these scripts.

## Parsel

Parsel<sup>1056</sup> is a CSS selector parser and specificity calculator, originally written by 2020 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.

---

1052. <https://www.thirdpartyweb.today/>

1053. <https://github.com/reworkcss/css>

1054. <https://discuss.httparchive.org/t/analyzing-stylesheets-with-a-js-based-parser/1683>

1055. <https://github.com/LeaVerou/rework-utils>

1056. <https://projects.verou.me/parsel/>

## 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.

### Planning

The 2021 Web Almanac kicked off in April 2021 with a call for contributors<sup>[1057](#)</sup>. We initialized the project with all 23 chapters from previous years and the community suggested additional topics that became two new chapters this year: Structured Data and WebAssembly.

As we stated in the inaugural year's Methodology:

*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 refined our author selection process<sup>[1058](#)</sup>:

- Previous authors were specifically discouraged from writing again to make room for different perspectives.
- Everyone endorsing 2021 authors were asked to be especially conscious not to nominate people who all look or think alike.
- 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.

### Analysis

In May and June 2021, data analysts worked with authors and peer reviewers to come up with a list of metrics that would need to be queried for each chapter. In some cases, custom metrics<sup>[1059](#)</sup>

<sup>1057</sup>. <https://github.com/HTTPArchive/almanac.httparchive.org/issues/2167>

<sup>1058</sup>. <https://github.com/HTTPArchive/almanac.httparchive.org/discussions/2165>

<sup>1059</sup>. <https://github.com/HTTPArchive/custom-metrics>

were created to fill gaps in our analytic capabilities.

Throughout July 2021, the HTTP Archive data pipeline crawled several million websites, gathering the metadata to be used in the Web Almanac. These results were post-processed and saved to BigQuery<sup>1060</sup>.

Being our third year, we were able to update and reuse the queries written by previous analysts. Still, there were many new metrics that needed to be written from scratch. You can browse all of the queries by year and chapter in our open source query repository<sup>1061</sup> 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 2021 edition of the Web Almanac is the third 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 2022 edition of the Web Almanac, please fill out our interest form<sup>1062</sup>. Let's work together to track the state of the web!

---

1060. <https://console.cloud.google.com/bigquery?p=httparchive&d=almanac&page=dataset>  
1061. <https://github.com/HTTPArchive/almanac.httparchive.org/tree/main/sql/2021>  
1062. <https://forms.gle/55uatdX9T3JZG2837>

# Appendix B

# Contributors



The Web Almanac has been made possible by the hard work of the web community. 121 people have volunteered countless hours in the planning, research, writing and production phases of the 2021 Web Almanac.



Abby Tsai

@AbbyTsai  
Developer



Alba Silvente Fuentes

@dawntraoz  
 Dawntraoz  
 <https://www.dawntraoz.com/>  
Reviewer



Adam Argyle

@argyleink  
 argyleink  
 <https://nerdy.dev>  
Reviewer



Alex Lakatos

@avolakatos  
 AlexLakatos  
 <http://alexlakatos.com/>  
Author



Addy Osmani

@addyosmani  
 addyosmani  
 <https://www.addyosmani.com>  
Reviewer



Alex Tait

@at\_fresh\_dev  
 alextait1  
 <https://atfreshsolutions.com>  
Author



Adriana Jara

tropicadri  
Reviewer



Alon Kochba

@alonkochba  
 alonkochba  
 [alonkochba.com](https://alonkochba.com)  
Author



Akshay Ranganath

akshay-ranganath  
Analyst and Reviewer



Alon Zakai

@kripken  
 kripken  
Reviewer



Alan Kent

@akent99  
 alankent  
 <https://alankent.me>  
Reviewer



Andrea Volpini

@cyberandy  
 cyberandy  
 <https://wordlift.io/blog/en/entity/andrea-volpini/>  
Author



**Andrey Lipattsev**  
✉ @AndreyLipattsev  
⌚ andreylipattsev  
Reviewer



**Carlo Piovesan**  
✉ @carlop54002226  
⌚ carlopi  
Reviewer



**André Cipriani Bandarra**  
✉ @andreban  
⌚ andreban  
Reviewer



**Cassey Lottman**  
⌚ clottman  
🌐 https://cassey.dev/  
Reviewer



**Andy Davies**  
✉ @AndyDavies  
⌚ andydavies  
🌐 http://andydavies.me/  
Reviewer



**Chris Lilley**  
✉ @svgeesus  
⌚ svgeesus  
🌐 https://svgeesus.us/  
Reviewer



**Artem Denysov**  
✉ @denar90\_  
⌚ denar90  
Analyst and Author



**Chris Sater**  
⌚ christophersater  
Reviewer



**Ashley Berman Hale**  
⌚ ashleyish  
Author



**Christian Liebel**  
✉ @christianliebel  
⌚ christianliebel  
🌐 https://christianliebel.com  
Author



**Barry Pollard**  
✉ @tunetheweb  
⌚ tunetheweb  
⌚ tunetheweb  
🌐 https://www.tunetheweb.com  
Analyst, Author, Developer, Editor, Project Lead, and Reviewer



**Dave Smart**  
✉ @davewsmart  
⌚ dwsmart  
🌐 https://tamethebots.com/  
Author



**Brian Kardell**  
✉ @briankardell  
⌚ bkardell  
🌐 https://bkardell.com  
Reviewer



**David Fox**  
✉ @theoboto  
⌚ foxdavidj  
🌐 https://www.lookzook.com  
Analyst, Project Lead, and Reviewer



**Caleb Queern**  
✉ @httpseheaders  
⌚ cqueern  
Reviewer



**Demian Renzulli**  
✉ @drenzulli  
⌚ demianrenzulli  
Analyst and Author



**Carlie Dixon**  
⌚ cdixon83  
Reviewer



**Dominic Lovell**  
✉ @dominiclovell  
⌚ dominiclovell  
Author

**Doug Sillars**

✉ dougsillars  
Analyst and Author

**Gertjan Franken**

✉ @GJFR\_  
✉ gjfr  
Analyst

**Edmond W. W. Chan**

✉ edmondwwchan  
Reviewer

**Gigi Rajani**

✉ GigiRajani  
Reviewer

**Eric A. Meyer**

✉ meyerweb  
✉ http://meyerweb.com/  
Author

**Greg Brimble**

✉ @gregbrimble  
✉ GregBrimble  
✉ https://gregbrimble.com/  
Analyst

**Eric Bailey**

✉ @ericwbailey  
✉ ericwbailey  
✉ https://ericwbailey.design/  
Reviewer

**Harry Roberts**

✉ @csswizardry  
✉ csswizardry  
✉ https://csswizardry.com/  
Reviewer

**Eric Portis**

✉ eeps  
✉ https://ericportis.com/  
Analyst and Author

**Hemanth HM**

✉ @gnumanth  
✉ hemanth  
✉ https://h3manth.com/  
Reviewer

**Estelle Weyl**

✉ @estellevv  
✉ estelle  
✉ http://standardista.com/  
Reviewer

**Ian Lurie**

✉ @ianlurie  
✉ wrttnwrd  
✉ https://www.ianlurie.com/  
Author

**Eugene Kliuchnikov**

✉ eustas  
Reviewer

**Ingvar Stepanyan**

✉ @RReverser  
✉ RReverser  
✉ https://rreverser.com/  
Analyst and Author

**Fili Wiese**

✉ @filiwiese  
✉ fili  
✉ filiwiese  
✉ https://filii.com/  
Reviewer

**Iulia Comsa**

✉ iulia-m-comsa  
✉ https://sites.google.com/view/  
iuliacomsa/  
Reviewer

**Gary Wilhelm**

✉ gwilhelm  
Author

**JR Oakes**

✉ @jroakes  
✉ jroakes  
Analyst

	<b>Jamie Indigo</b> Twitter: @Jammer_Volts GitHub: fellowhuman1101 Website: https://not-a-robot.com/ Author and Reviewer		<b>Jono Alderson</b> Twitter: @jonoalderson GitHub: jonoalderson Website: https://www.jonoalderson.com/ Author
	<b>Jarno van Driel</b> Twitter: @JarnoVanDriel GitHub: jvandriel LinkedIn: jarno-van-driel-36a47075 Editor		<b>Julia Yang</b> Twitter: @Jules_Yang GitHub: jzyang LinkedIn: yangzhe Editor and Reviewer
	<b>Jarrod Overson</b> Twitter: @jsoverson GitHub: http://jarrodroverson.com/ Reviewer		<b>Jyrki Alakuijala</b> Twitter: @jyzg GitHub: jyrkilalakuijala Author
	<b>Jasmine Drudge-Willson</b> Twitter: @jasminedwillson GitHub: JasmineDWillson Editor		<b>Kai Hollberg</b> Twitter: @schweinepriestr GitHub: Schweinepriester Reviewer
	<b>Jeff Posnick</b> Twitter: @jeffposnick GitHub: jeffposnick Website: https://jeffy.info/ Reviewer		<b>Katriel Paige</b> GitHub: kachiden Website: https://www.flowerstorm.tech/ Author
	<b>Jens Oliver Meiert</b> Twitter: @j9t GitHub: j9t Website: https://meiert.com/en/ Reviewer		<b>Kevin Farrugia</b> Twitter: @imkevdev GitHub: kevinfarrugia Website: https://imkev.dev/ Analyst, Author, and Reviewer
	<b>Jess Peck</b> Twitter: @jessthebp GitHub: jessthebp Website: https://jessbpeck.com/ Analyst		<b>Koen Van den Wijngaert</b> Twitter: @vdwijngaert GitHub: vdwijngaert Website: https://www.neok.be/ Reviewer
	<b>Jessica Nicolet</b> Twitter: @jessica_nicolet GitHub: jessnicolet Website: https://www.jessicanicolet.com/ Author		<b>Lea Verou</b> Twitter: @leaverou GitHub: LeaVerou Website: https://lea.verou.me/ Reviewer
	<b>John Teague</b> Twitter: @jteag GitHub: logicalphase Website: https://gemservers.com/ Author and Reviewer		<b>Leonardo Zizzamia</b> Twitter: @Zizzamia GitHub: Zizzamia Website: https://twitter.com/zizzamia Author

**Lode Vandevenne**

lvandeve  
Author

**Moritz Firsching**

mo271  
Author

**Lucas Gonçalves**

lucasbona05  
Developer

**Mukesh Jat**

mukeshjat  
Translator

**Manuel Garcia**

@corrosion\_pt  
 soulcorrosion  
 manuel-garcia-12b6928  
<https://farfetchtechblog.com/en/blog/authors/manuel-garcia/>  
Reviewer

**Navaneeth Krishna**

@Navanee55755217  
 Navaneeth-akam  
Author and Reviewer

**Matteo Große-Kampmann**

awareseven  
Reviewer

**Nikita Dubko**

@dark\_mefody  
 MeFoDy  
 https://mefody.dev/  
Translator

**Maud Nalpas**

maudnals  
Reviewer

**Nishu Goel**

@TheNishuGoel  
 NishuGoel  
 http://unravelweb.dev/  
Author

**Max Ostapenko**

@themax\_o  
 max-ostapenko  
 https://maxostapenko.com  
Analyst

**Nitin Pasumarty**

Nithanaroy  
 nitinpasumarty  
 https://nithanaroy.medium.com/  
Analyst

**Maxim Salnikov**

@webmaxru  
 webmaxru  
Reviewer

**Nurullah Demir**

@nrllah  
 nrllh  
 https://internet-sicherheit.de  
Author

**Michelle O'Connor**

Designer

**Olu Niyi-Awosusi**

@oluoluoxenfree  
 oluoluoxenfree  
 https://olu.online/  
Author

**Minko Gechev**

@mgechev  
 mgechev  
 https://blog.mgechev.com/  
Reviewer

**Pankaj Parkar**

@pankajparkar  
 pankajparkar  
 https://medium.com/@pankajparkar  
Analyst, Editor, and Reviewer



**Pascal Schilp**

✉ thepassple  
Reviewer



**Robin Marx**

✉ @programmingart  
✉ rmarx  
Reviewer



**Patrick Hulce**

✉ @patrickhulce  
✉ patrickhulce  
✉ http://patrickhulce.com  
Reviewer



**Rockey Nebhwani**

✉ @rnebhwanie  
✉ rockeynebhwanie  
✉ rockeynebhwanie  
Reviewer



**Patrick Stox**

✉ @patrickstox  
✉ patrickstox  
✉ https://patrickstox.com  
Author



**Rory Hewitt**

✉ @roryhewitt3  
✉ roryhewitt  
✉ roryhewitt  
✉ https://romche.com  
Reviewer



**Paul Calvano**

✉ @paulcalvano  
✉ paulcalvano  
✉ https://paulcalvano.com  
Analyst and Project Lead



**Ruth Everett**

✉ @rvtheverett  
✉ rvth  
Analyst



**Phil Barker**

✉ @philbarker  
✉ philbarker  
✉ https://blogs.pjjk.net/phil/  
Reviewer



**Sakae Kotaro**

✉ @beltway7  
✉ ksakae1216  
✉ https://ksakae1216.com/  
Translator



**Rajiv Ramnath**

✉ rrajiv  
✉ rajivramnath  
Analyst



**Samar Panda**

✉ samarpanda  
Reviewer



**Rebecca Holmlund**

✉ RMHolmlund  
Reviewer



**Saptak Sengupta**

✉ @Saptak013  
✉ saptaks  
✉ https://saptaks.website/  
Author and Developer



**Rick Viscomi**

✉ @rick\_viscomi  
✉ rviscomi  
Analyst, Editor, Project Lead, and  
Reviewer



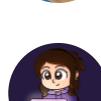
**Scott Davis**

✉ scottdavis99  
Author



**Rob Teitelman**

✉ @teitelmanrob  
✉ SeoRobt  
✉ https://www.paulteitelman.com/  
Reviewer



**Shaina Hantsis**

✉ shantsis  
Designer, Editor, and Reviewer

**Shilpa Raghunathan**

[boosef](#)  
Reviewer

**Tom Robertshaw**

[@bobbyshaw](#)  
 [bobbyshaw](#)  
 [tomrobertshaw](#)  
 <https://www.space48.com>  
Author

**Shuvam Manna**

[@shuvam360](#)  
 [GeekBoySupreme](#)  
 <https://shuvam.xyz>  
Author and Designer

**Tom Van Goethem**

[@tomvangoethem](#)  
 [tomvangoethem](#)  
Author

**Sia Karamalegos**

[@TheGreenGreek](#)  
 [siakaramalegos](#)  
 [karamalegos](#)  
 <https://sia.codes>  
Analyst, Author, and Reviewer

**Tomek Rudzki**

[@TomekRudzki](#)  
 [Tomek3c](#)  
 <https://tomekseo.com/>  
Author

**Simon Hearne**

[@simonhearne](#)  
 [simonhearne](#)  
 <https://simonhearne.com>  
Reviewer

**Tosin Arasi**

[tosinarasi](#)  
Analyst

**Tamas Piros**

[@tpiros](#)  
 [tpiros](#)  
 <https://tamas.io>  
Reviewer

**Victor Le Pochat**

[@VictorLePochat](#)  
 [VictorLeP](#)  
 [victor-le-pochat](#)  
 <https://lepoche.at>  
Analyst, Author, and Translator

**Thom Krupa**

[@thomkrupa](#)  
 [thomkrupa](#)  
 <https://www.thomkrupa.com/>  
Reviewer

**Weston Ruter**

[@westonruter](#)  
 [westonruter](#)  
 <https://weston.ruter.net/>  
Reviewer

**Thomas Fischbacher**

[fischbacher](#)  
Reviewer

**Wilhelm Willie**

[WilhelmWillie](#)  
Reviewer

**Thomas Steiner**

[tomayac](#)  
 <https://blog.tomayac.com/>  
Analyst and Reviewer

**Yana Dimova**

[ydimova](#)  
Author

**Timur Kartashov**

[@krutoi\\_paren](#)  
 [kartashovio](#)  
 <https://kartashov.io/>  
Translator

**Yusuf Seyhan**

[@yuseyhan](#)  
 [yuseyhan](#)  
 <https://webpen.de/>  
Designer



**Ziemek Bućko**

[ziemek-bucko](#)

Reviewer