

# 2020

# Web Almanac

---

HTTP Archive's annual  
**state of the web** report





# Table of Contents

## Introduction

Foreword .....	iii
----------------	-----

## Part I. Page Content

Chapter 1: CSS .....	.1
Chapter 2: JavaScript .....	51
Chapter 3: Markup .....	53
Chapter 4: Fonts .....	81
Chapter 5: Media .....	99
Chapter 6: Third Parties .....	101

## Part II. User Experience

Chapter 7: SEO .....	103
Chapter 8: Accessibility .....	141
Chapter 9: Performance .....	143
Chapter 10: Privacy .....	165
Chapter 11: Security .....	175
Chapter 12: Mobile Web .....	177
Chapter 13: Capabilities .....	179
Chapter 14: PWA .....	197

## Part III. Content Publishing

Chapter 15: CMS .....	213
Chapter 16: Ecommerce .....	233
Chapter 17: Jamstack .....	235

## Part IV. Content Distribution

Chapter 18: Page Weight .....	249
Chapter 19: Compression .....	251

Chapter 20: Caching .....	261
Chapter 21: Resource Hints .....	297
Chapter 22: HTTP/2 .....	313

## Appendices

Methodology.....	335
Contributors.....	345

# Foreword

Coming soon!



## Part I Chapter 1

# CSS [UNEDITED]



*Written by Lea Verou, Chris Lilley, and Rachel Andrew*

*Reviewed by Fantasai, Miriam Suzanne, Jens Oliver Meiert, Catalin Rosu, and Andy Bell*

*Analyzed by Tony McCreath*

# Introduction

## Methodology

## Usage



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



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



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

## Selectors and the cascade

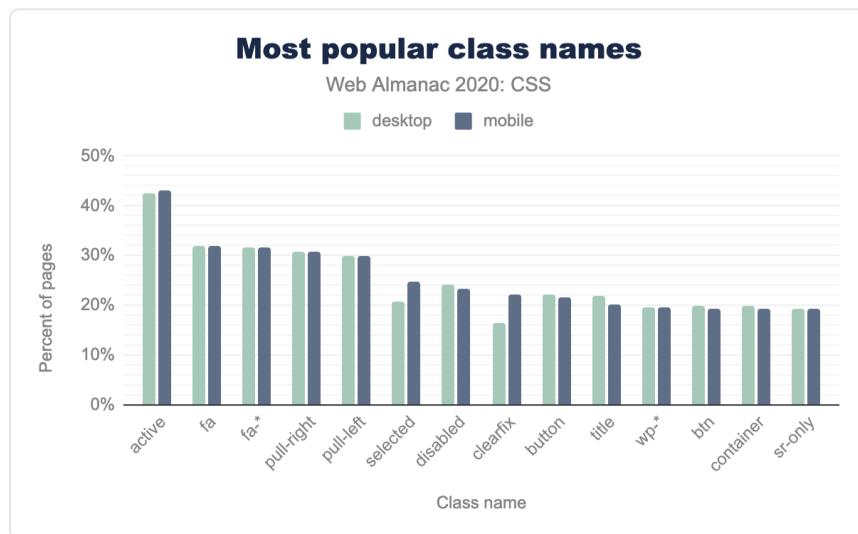


Figure 1.4. The most popular class names by the percent of pages.

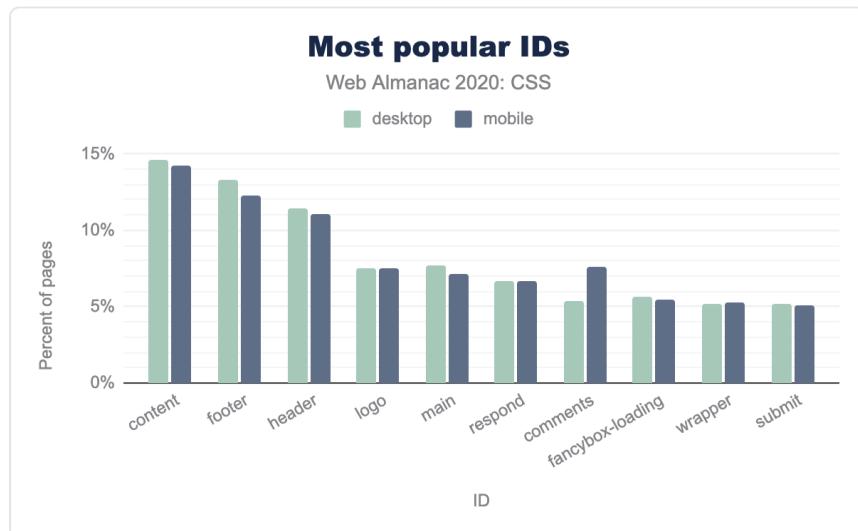


Figure 1.5. The most popular IDs by the percent of pages.

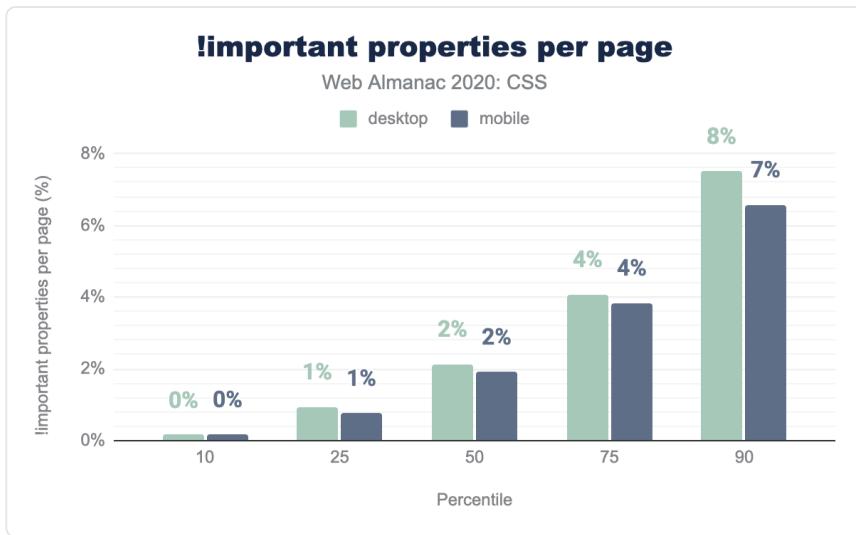


Figure 1.6. Distribution of the percent of !important properties per page.

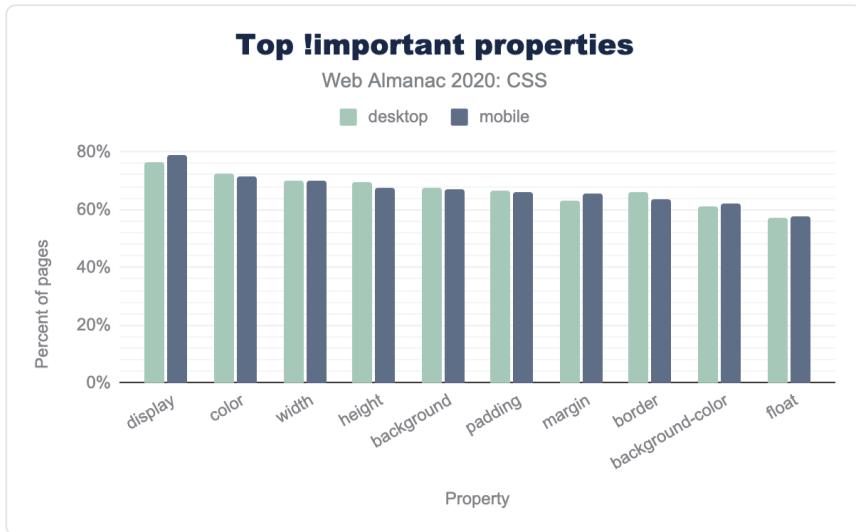


Figure 1.7. The top !important properties by the percent of pages.

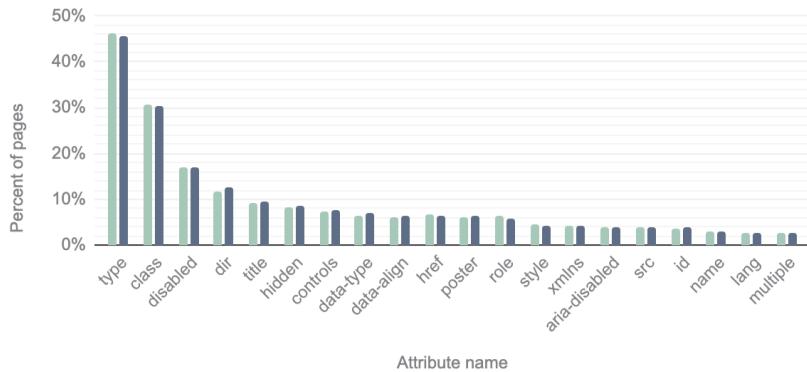
<b>Percentile</b>	<b>Desktop</b>	<b>Mobile</b>
10	0,1,0	0,1,0
25	0,2,0	0,1,2
50	0,2,0	0,2,0
75	0,2,0	0,2,0
90	0,3,0	0,3,0

*Figure 1.8. Distribution of the median specificity per page.*

## Most popular attribute selectors

Web Almanac 2020: CSS

desktop  
 mobile

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

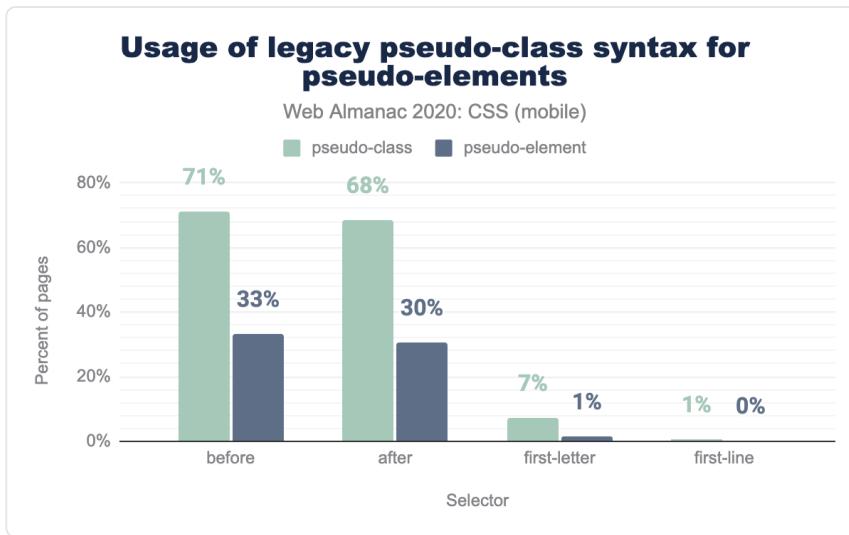


Figure 1.10. Usage of legacy `:pseudo-class` syntax for `::pseudo-elements` as a percent of mobile pages.

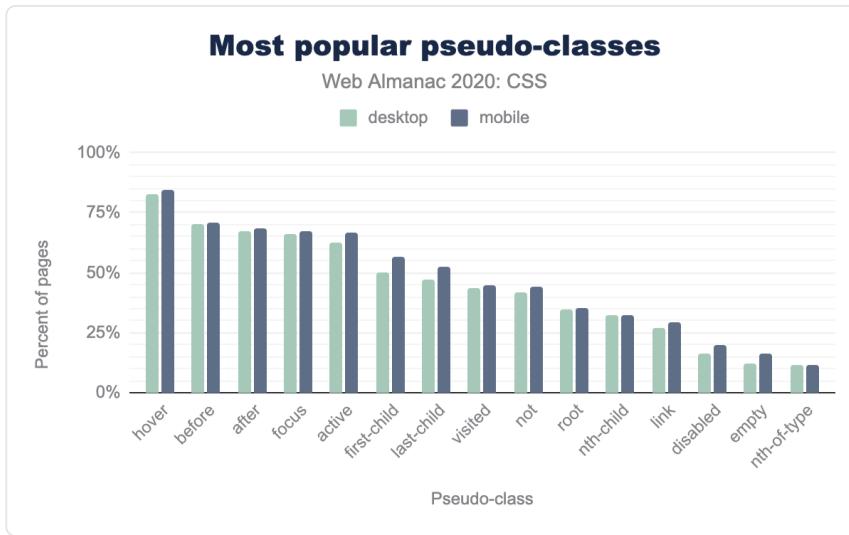


Figure 1.11. The most popular pseudo-classes as a percent of pages.

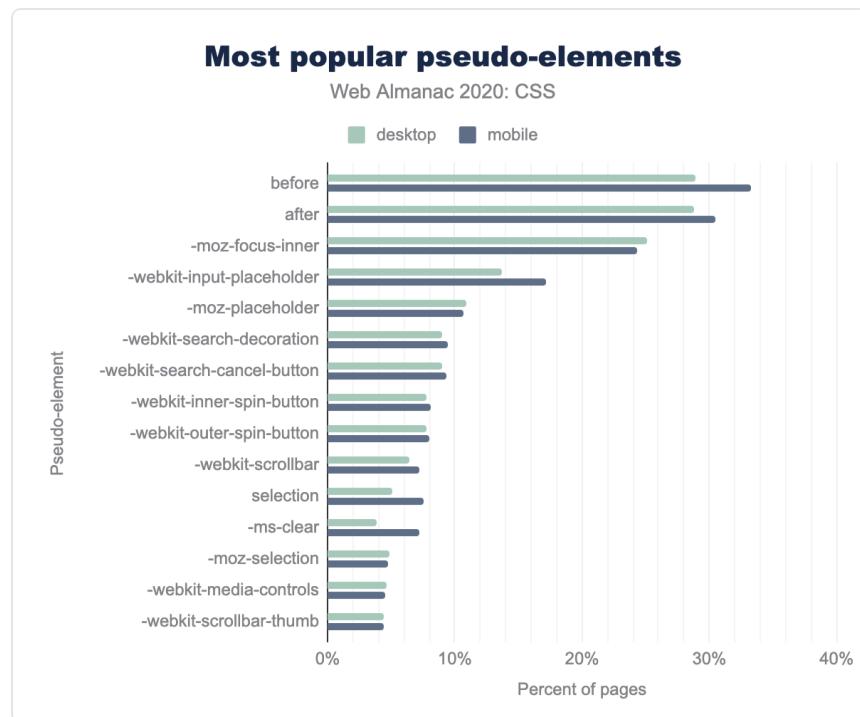


Figure 1.12. The most popular pseudo-elements as a percent of pages.

## Values and units

### Lengths

72.58%

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

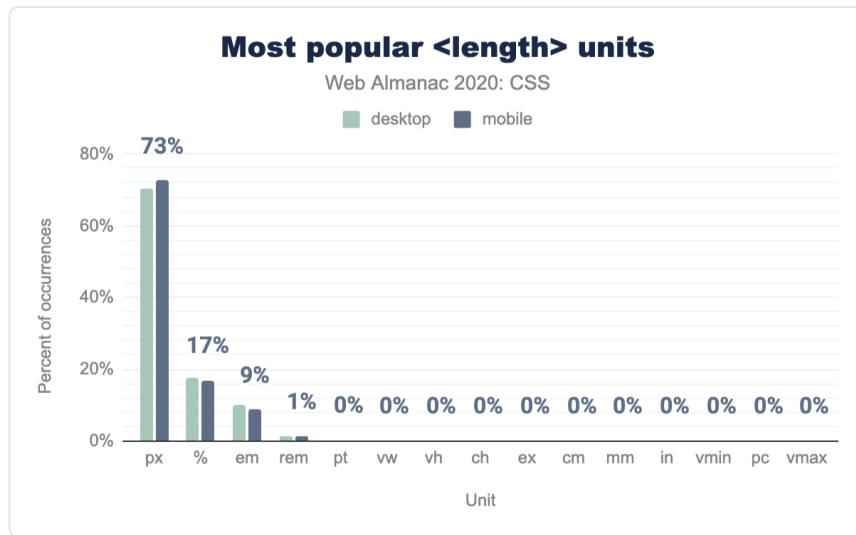


Figure 1.14. The most popular <length> units as a percent of occurrences.

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

Figure 1.15. Unit usage by property.

## Most popular font-relative units

Web Almanac 2020: CSS

desktop mobile

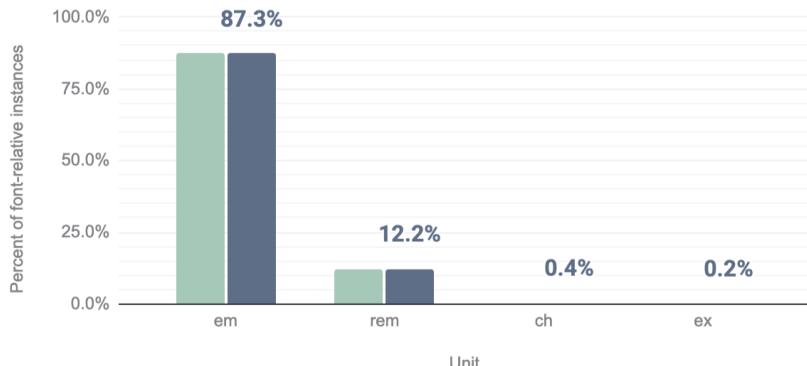


Figure 1.16. Relative popularity of font-based units other than `px` as a percent of occurrences.

## 0 lengths by unit

Web Almanac 2020: CSS

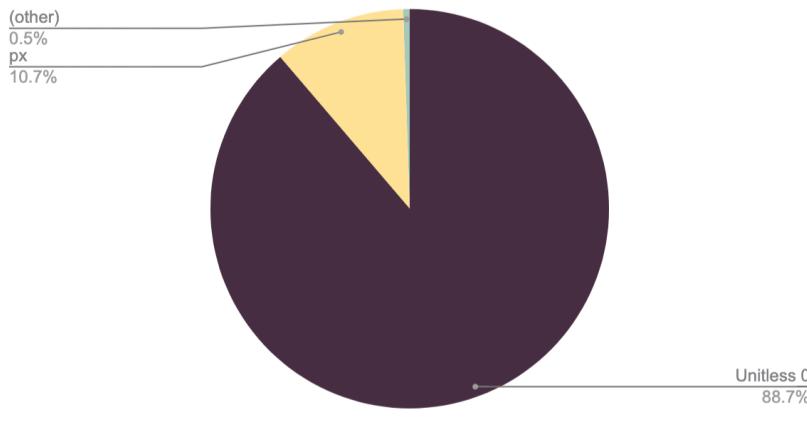


Figure 1.17. Relative popularity of 0 lengths by unit as a percent of occurrences on mobile pages.

## Calculations

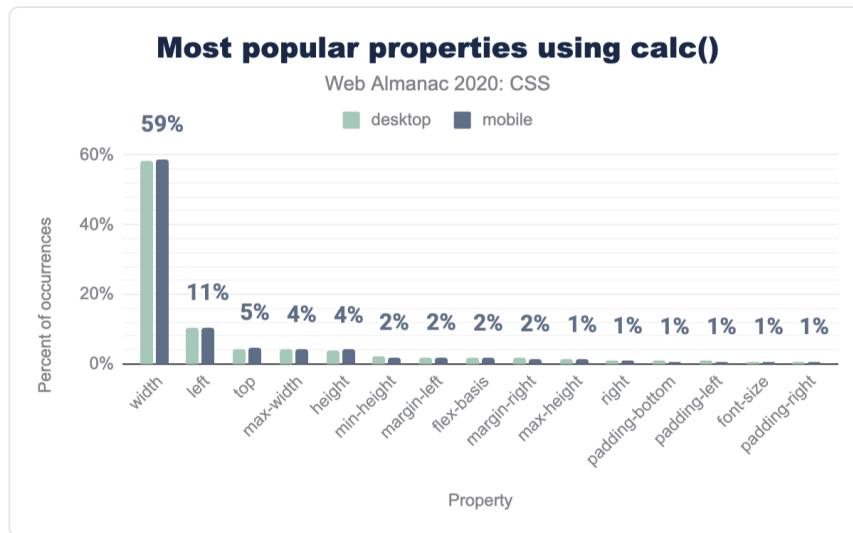


Figure 1.18. Relative popularity of properties that use `calc()` as a percent of occurrences.

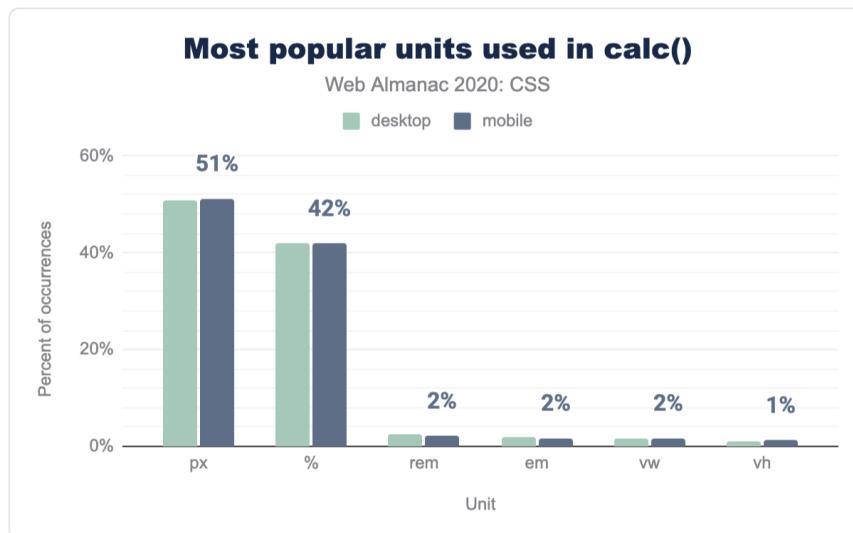


Figure 1.19. Relative popularity of units that use `calc()` as a percent of occurrences.

## Most popular operators used in calc()

Web Almanac 2020: CSS

desktop mobile

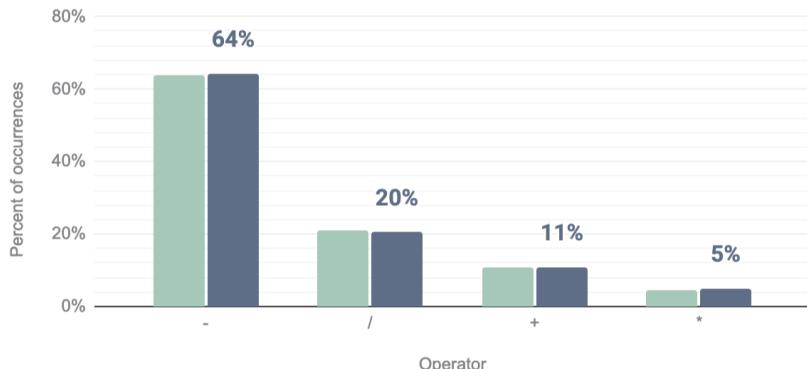


Figure 1.20. Relative popularity of operators that use `calc()` as a percent of occurrences.

## Number of units per calc() occurrence

Web Almanac 2020: CSS

desktop mobile

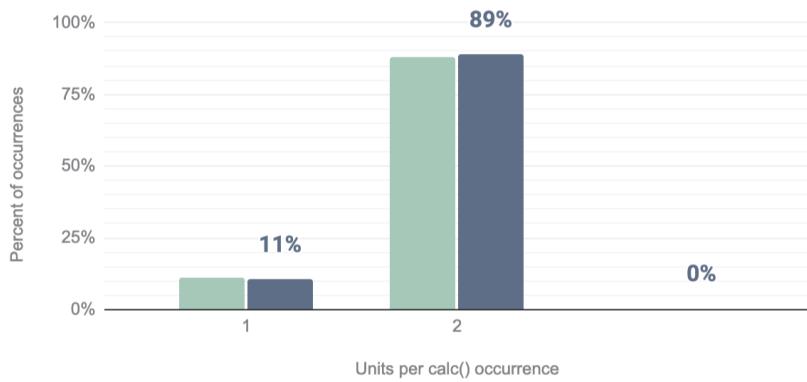


Figure 1.21. Distribution of the number of units per `calc()` occurrence.

## Global keywords and `all`

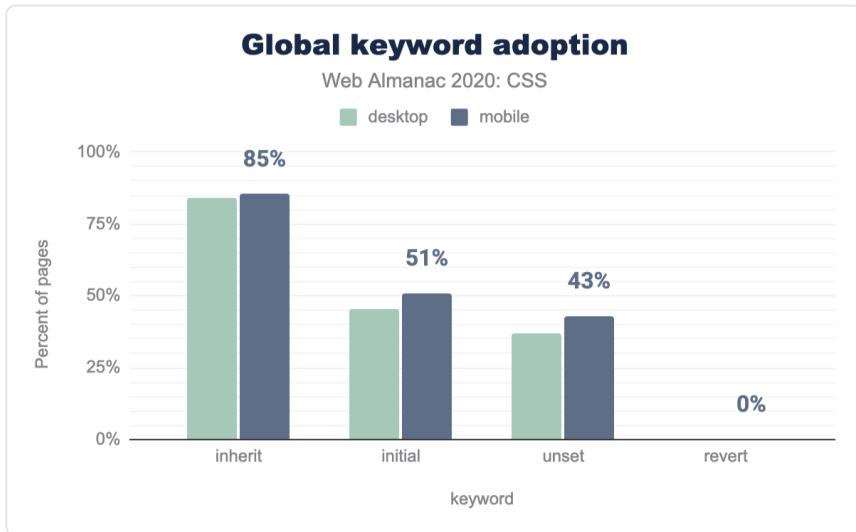


Figure 1.22. Adoption of global keywords as a percent of pages.

## Color

### Most popular color formats

Web Almanac 2020: CSS

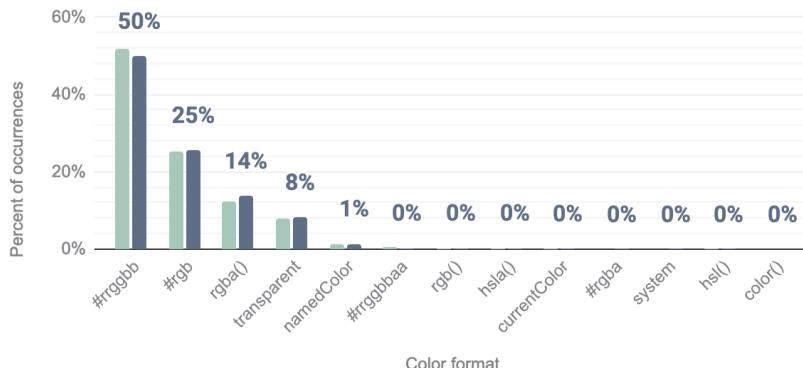
■ desktop    ■ mobile


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

### Color formats by alpha support

Web Almanac 2020: CSS (mobile)

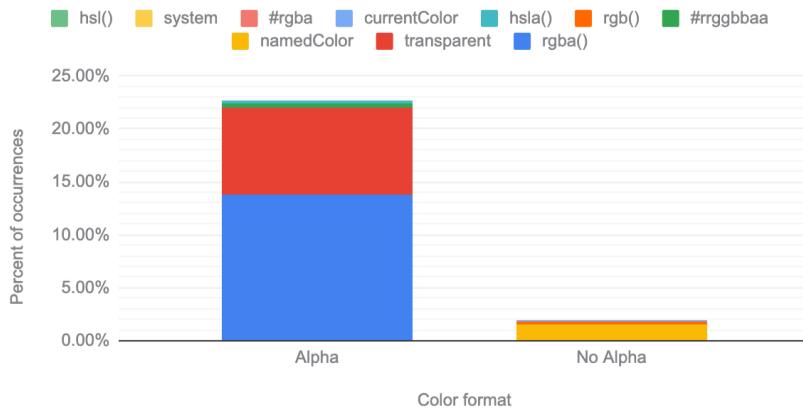


Figure 1.24. Relative popularity of color formats grouped by alpha support as a percent of occurrences on mobile pages (excluding `#rrgbb` and `#rgb`).

<b>Keyword</b>	<b>Desktop</b>	<b>Mobile</b>
transparent	84.04%	83.51%
white	6.82%	7.34%
black	2.32%	2.42%
red	2.03%	2.01%
currentColor	1.43%	1.43%
gray	0.75%	0.79%
silver	0.66%	0.58%
grey	0.35%	0.31%
green	0.36%	0.30%
magenta	0.00%	0.13%
blue	0.16%	0.13%
whitesmoke	0.17%	0.12%
lightgray	0.06%	0.11%
orange	0.12%	0.10%
lightgrey	0.04%	0.10%
yellow	0.08%	0.06%
Highlight	0.01%	0.04%
gold	0.04%	0.04%
pink	0.03%	0.03%
teal	0.03%	0.02%

Figure 1.25. Relative popularity of color keywords as a percent of occurrences.

<b>sRGB</b>		<i>display-p3</i>	<b>ΔE2000</b>	<b>In gamut</b>
<code>rgba(0,0,0,1)</code>		<code>color(display-p3 0 0 0 / 1)</code>	0.000	true
<code>rgba(255,255,255,1)</code>		<code>color(display-p3 1 1 1 / 1)</code>	0.015	false
<code>rgba(200,200,200,1)</code>		<code>color(display-p3 0.78 0.78 0.78 / 1)</code>	0.274	true
<code>rgba(121,127,132,1)</code>		<code>color(display-p3 0.48 0.50 0.52 / 1)</code>	0.391	true
<code>rgba(255,205,63,1)</code>		<code>color(display-p3 1 0.80 0.25 / 1)</code>	3.880	false
<code>rgba(241,174,50,1)</code>		<code>color(display-p3 0.95 0.68 0.17 / 1)</code>	4.701	false
<code>rgba(245,181,40,1)</code>		<code>color(display-p3 0.96 0.71 0.16 / 1)</code>	4.218	false
<code>rgb(147, 83, 255)</code>		<code>color(display-p3 0.58 0.33 1 / 1)</code>	2.143	false
<code>rgba(120,0,255,1)</code>		<code>color(display-p3 0.47 0 1 / 1)</code>	1.933	false
<code>rgba(75,3,161,1)</code>		<code>color(display-p3 0.29 0.01 0.63 / 1)</code>	1.321	false
<code>rgba(255,0,0,0.85)</code>		<code>color(display-p3 1 0 0 / 0.85)</code>	7.115	false
<code>rgba(84,64,135,1)</code>		<code>color(display-p3 0.33 0.25 0.53 / 1)</code>	1.326	true
<code>rgba(131,103,201,1)</code>		<code>color(display-p3 0.51 0.40 0.78 / 1)</code>	1.348	true
<code>rgba(68,185,208,1)</code>		<code>color(display-p3 0.27 0.75 0.82 / 1)</code>	5.591	false
<code>rgb(255,0,72)</code>		<code>color(display-p3 1 0 0.2823 / 1)</code>	3.529	false

<b>sRGB</b>	<b>display-p3</b>	<b>ΔE2000</b>	<b>In gamut</b>
#ffffc00	[Yellow] color(display-p3 1 0.9882 0)	5.012	false
#6d3bff	[Purple] color(display-p3 .427 .231 1)	1.584	false
#03d658	[Green] color(display-p3 .012 .839 .345)	4.958	false
#ff3900	[Red] color(display-p3 1 .224 0)	7.140	false
#7cf8b3	[Cyan] color(display-p3 .486 .973 .702)	4.284	true
#f8f8f8	[White] color(display-p3 .973 .973 .973)	0.028	true
#e3f5fd	[Light Blue] color(display-p3 .875 .945 .976)	1.918	true

Figure 1.26. The fallback sRGB colors and display-p3 colors.  
 #e74832       #905882  
 #6d3bff       #7cf8b3  
 #ffffc00       #f8f8f8  
 #e3f5fd      A color difference ( $\Delta E_{2000}$ ) of 1 is barely visible, while 5 is clearly distinct.  
 #e74832       #905882  
 #6d3bff       #7cf8b3  
 #ffffc00       #f8f8f8  
 #e3f5fd      .196078431 / 1 )      true

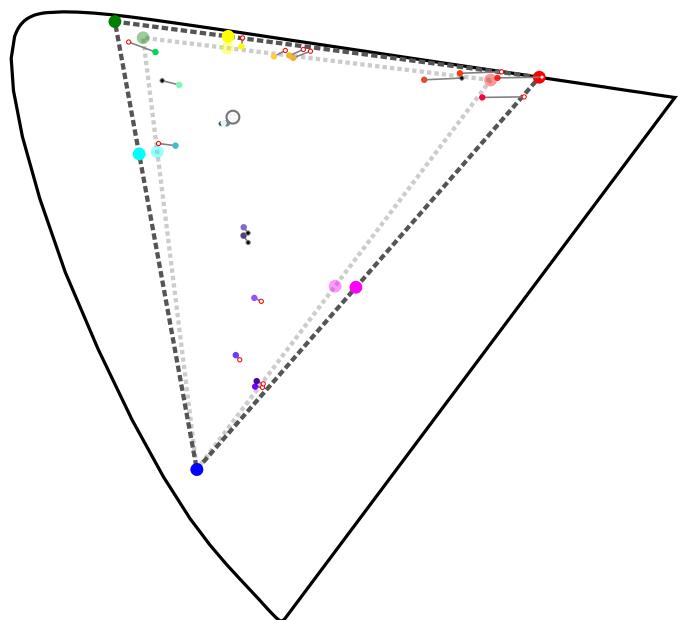


Figure 1.27. 1976  $u'v'$  diagram of the chromaticity of colors.

## Gradients

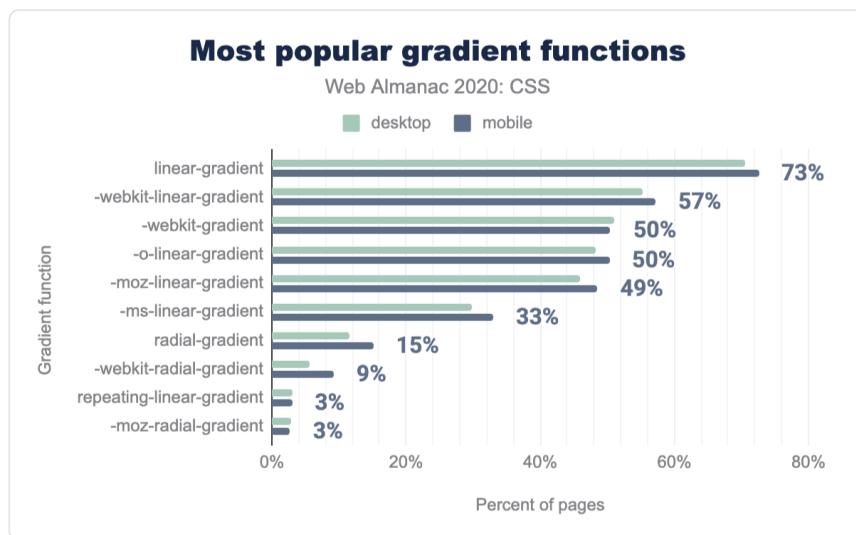


Figure 1.28. The most popular gradient functions as a percent of pages.

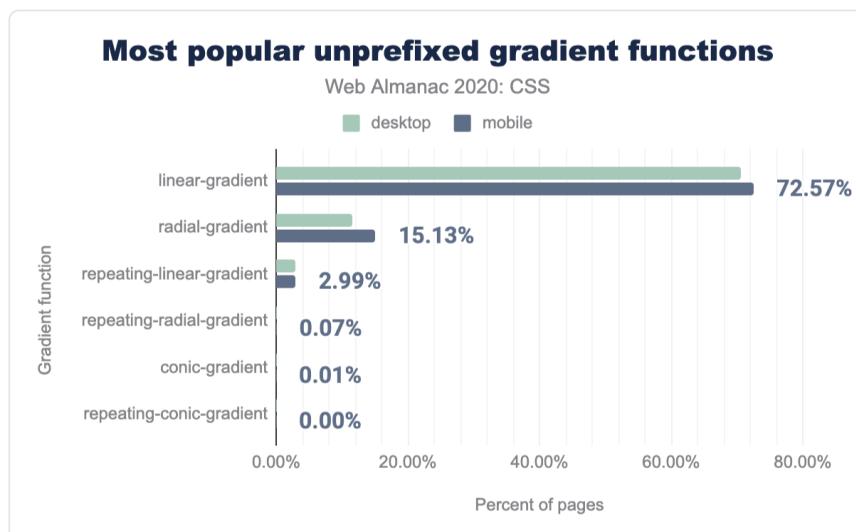


Figure 1.29. The most popular gradient functions as a percent of pages, omitting vendor prefixes.



Figure 1.30. The gradient with the most color stops, 646.

## Layout

### Flexbox and Grid adoption

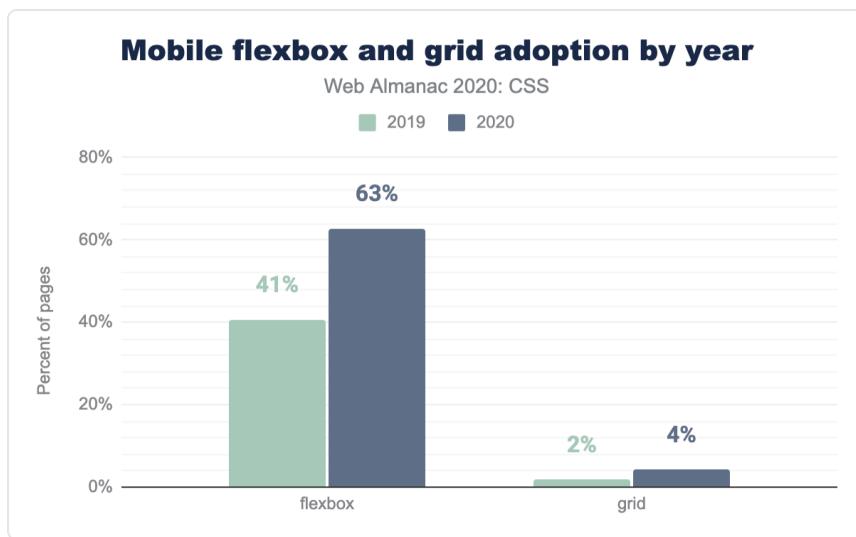


Figure 1.31. Adoption of flexbox and grid by year as a percent of mobile pages.

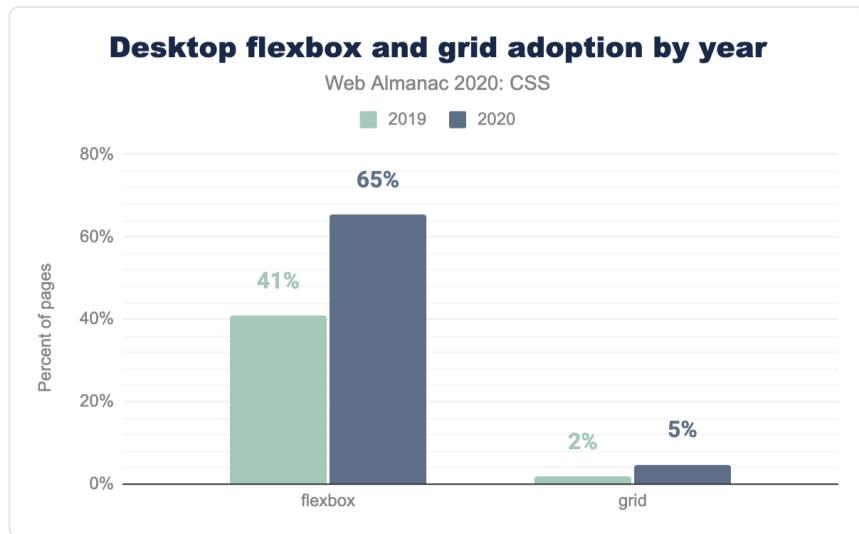


Figure 1.32. Adoption of flexbox and grid by year as a percent of desktop pages.

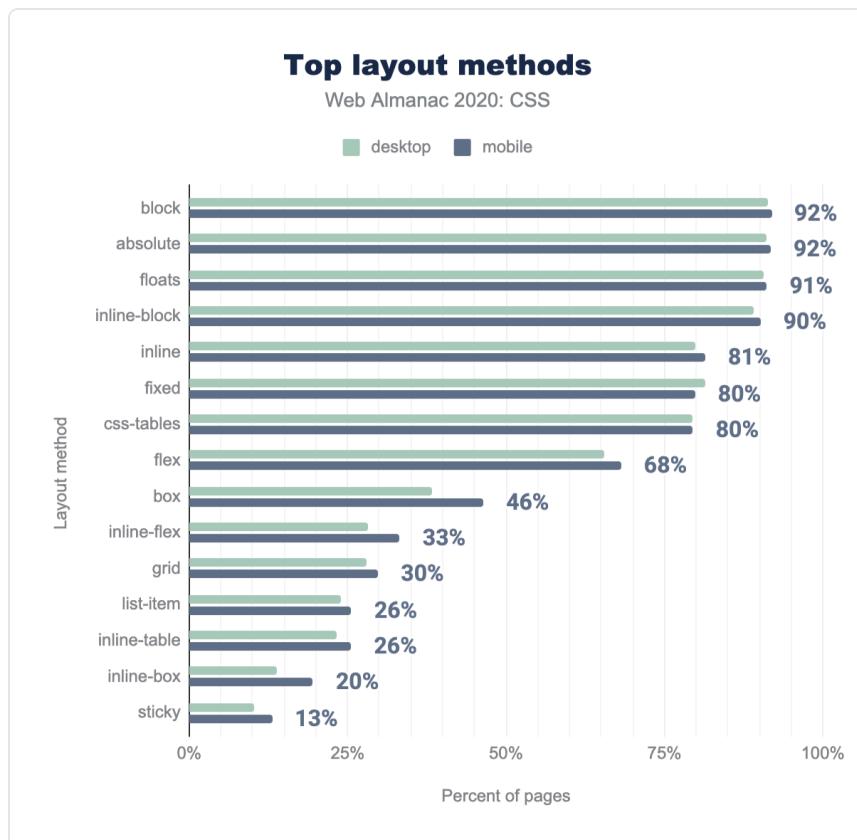


Figure 1.33. Adoption of layout methods as a percent of pages.

## Usage of different Grid layout techniques

### Multiple-column layout

#### Box sizing

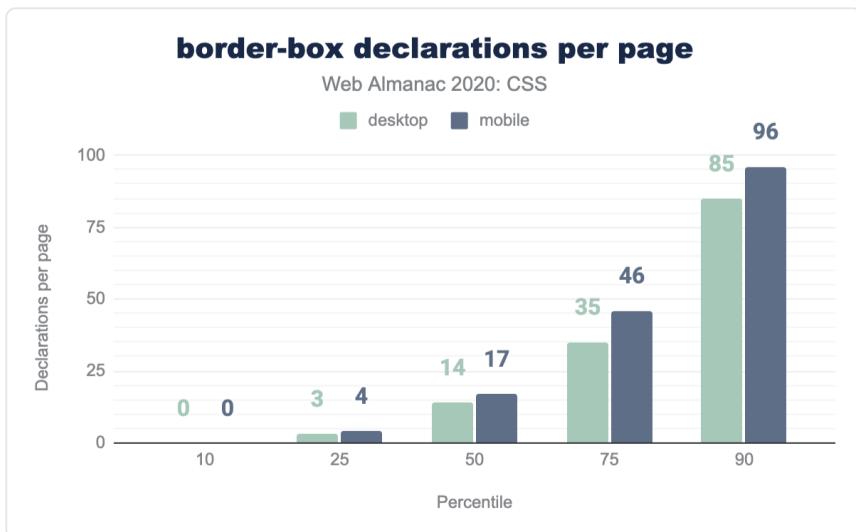


Figure 1.34. Distribution of the number of border-box declarations per page.

## Transitions and animations

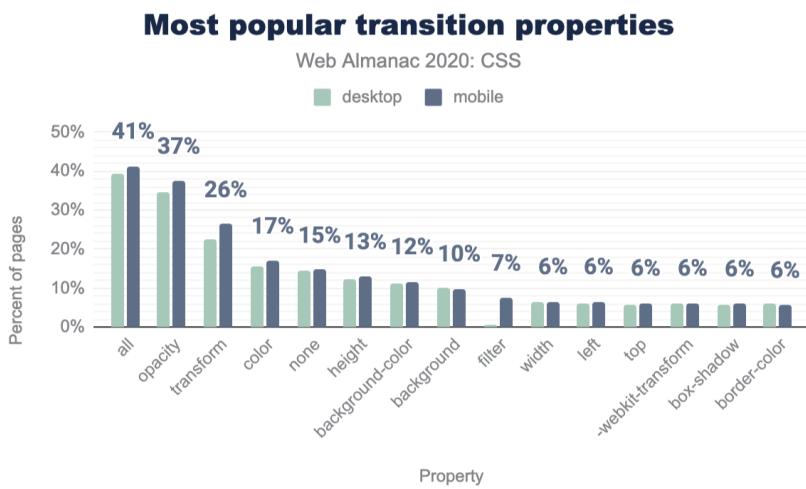


Figure 1.35. Adoption of transition properties as a percent of pages.

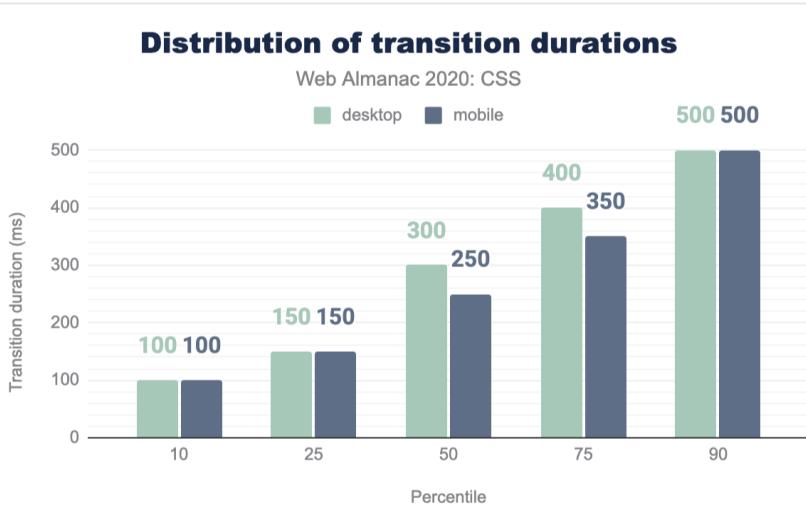


Figure 1.36. Distribution of transition durations.

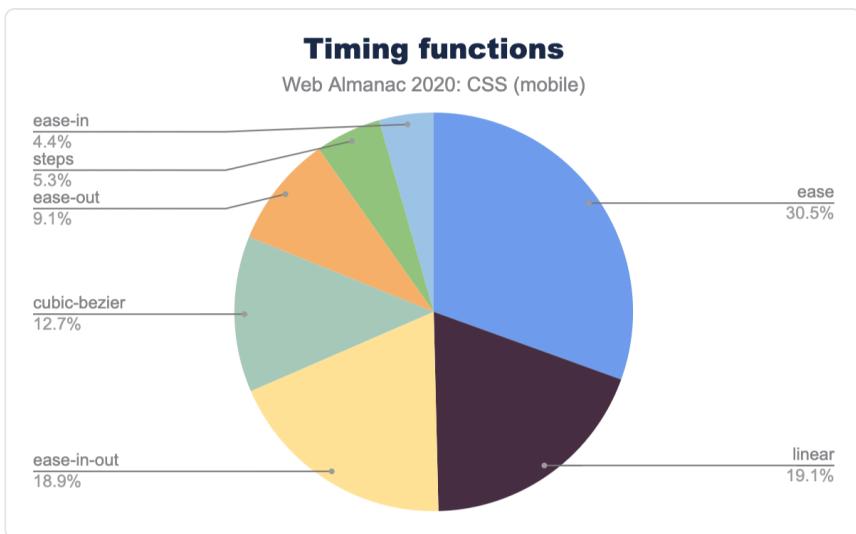


Figure 1.37. Relative popularity of timing functions as a percent of occurrences on mobile pages.

# Visual effects

## Blend modes

## Filters

## Masks

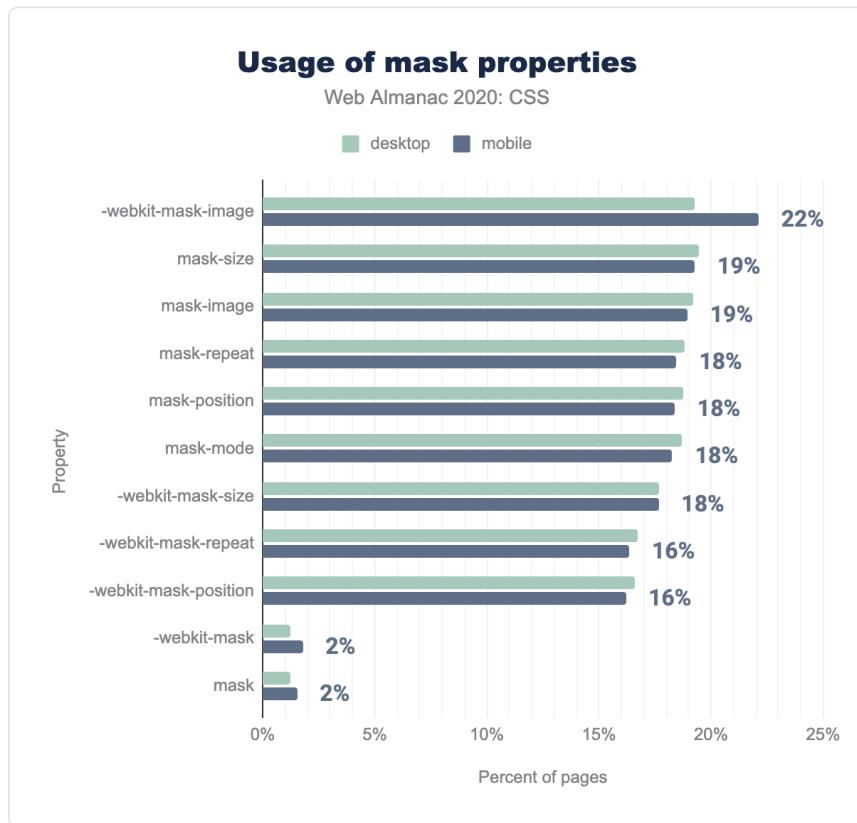


Figure 1.38. Relative popularity of animation name categories as a percent of occurrences.

## Clipping paths

# Responsive design

Which media features are people using?

## Common breakpoints

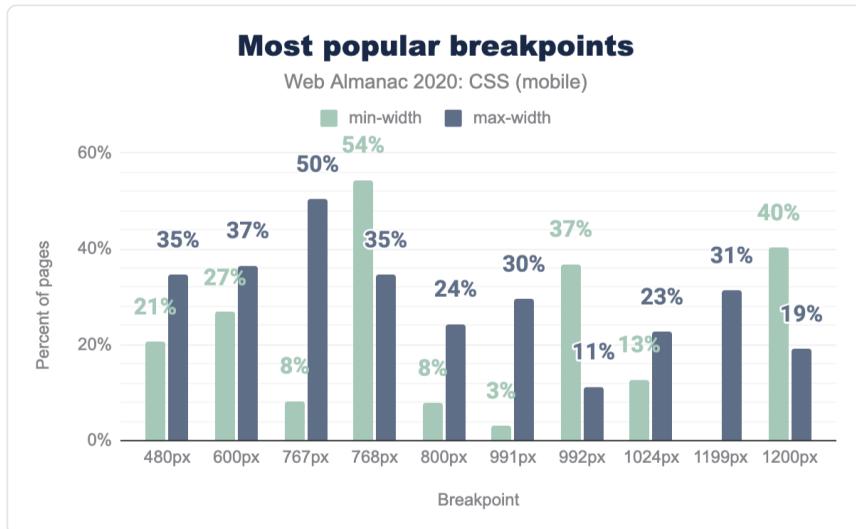


Figure 1.39. The most popular breakpoints by min-width and max-width as a percent of mobile pages.

## Most popular properties used in media queries

Web Almanac 2020: CSS

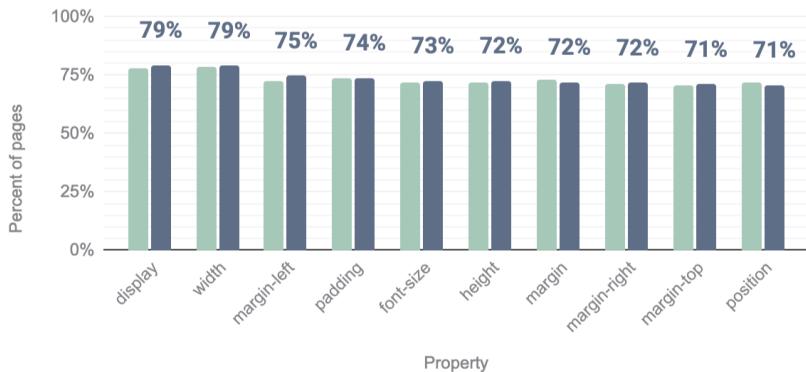
■ desktop   ■ mobile

Figure 1.40. The most popular properties used in media queries as a percent of pages.

## Properties used inside media queries

### Custom properties

#### Naming

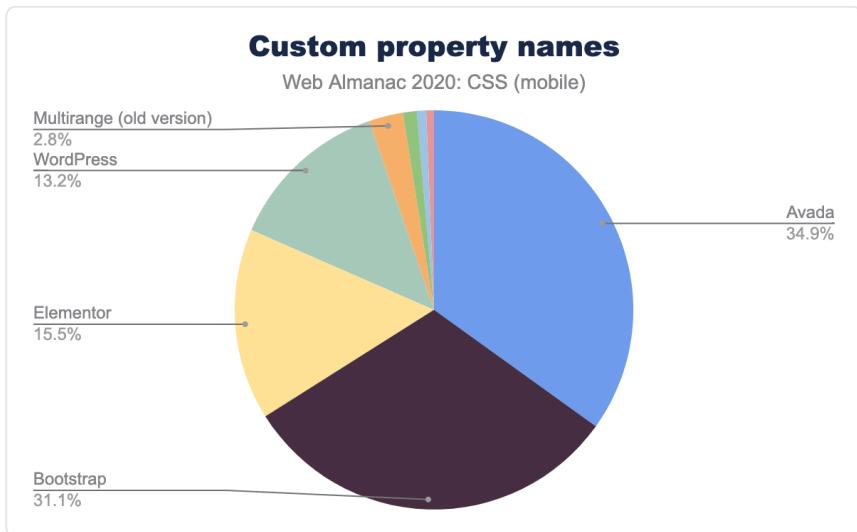


Figure 1.41. Relative popularity of custom property names per software entity as a percent of occurrences on mobile pages.

## Usage by type

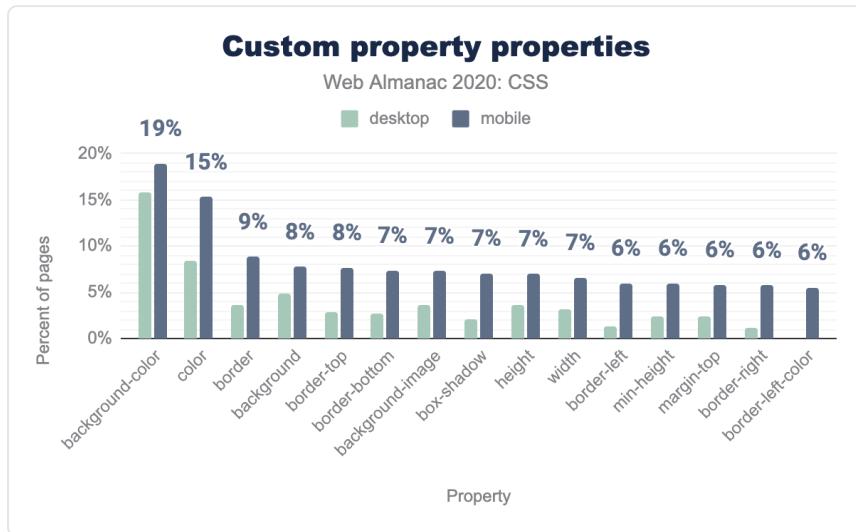


Figure 1.42. The most popular property names used with custom properties as a percent of pages.

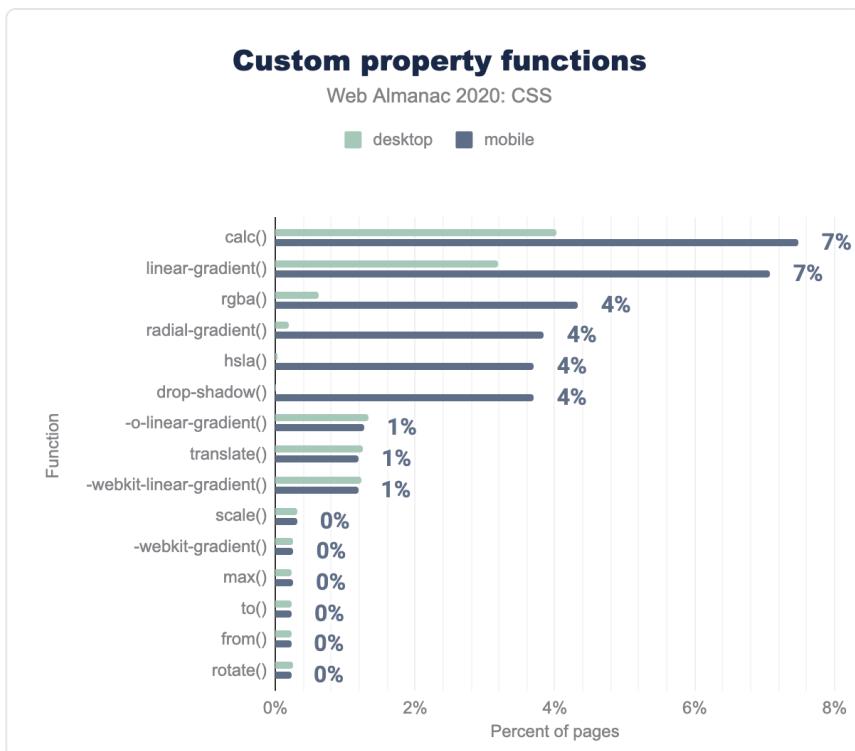


Figure 1.43. The most popular function names used with custom properties as a percent of pages.

## Complexity

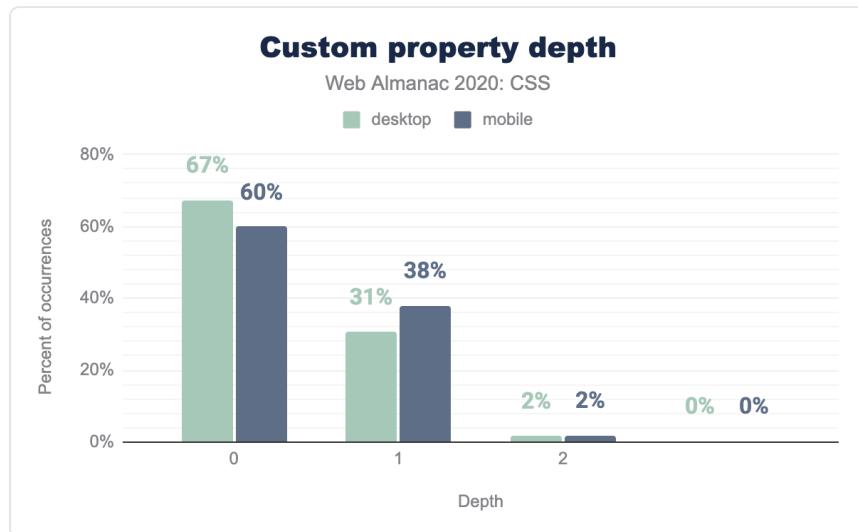


Figure 1.44. Distribution of depths of custom properties as a percent of occurrences.

## CSS and JS

### Houdini

#### CSS-in-JS

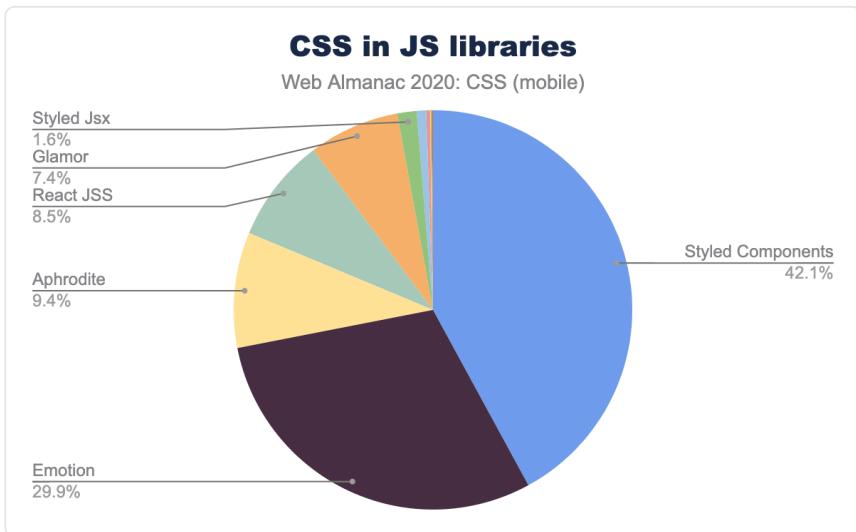


Figure 1.45. Relative popularity of CSS-in-JS libraries as a percent of occurrences on mobile pages.

# Internationalization

## Direction

### Logical vs physical properties

## Browser support

### Vendor prefixes

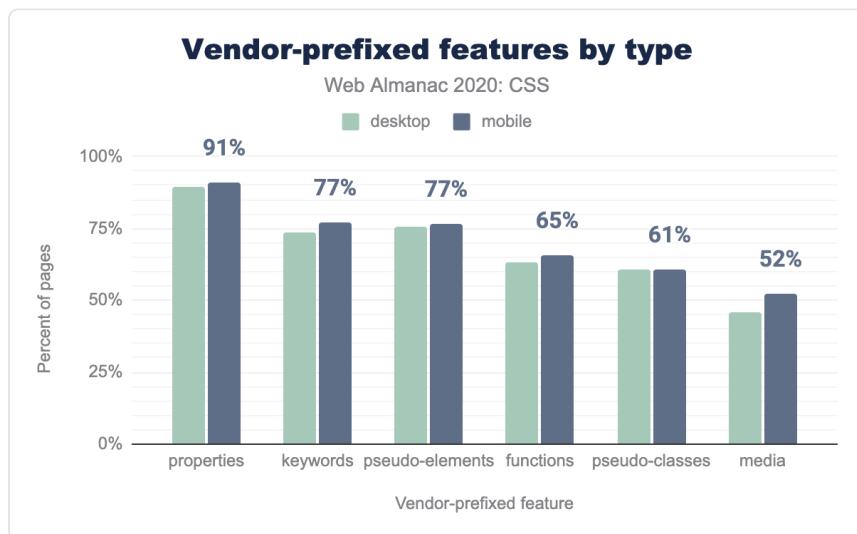


Figure 1.46. The most popular vendor-prefixed features by type as a percent of pages.

91.05%

Figure 1.47. Percent of mobile pages using any vendor prefixed feature.

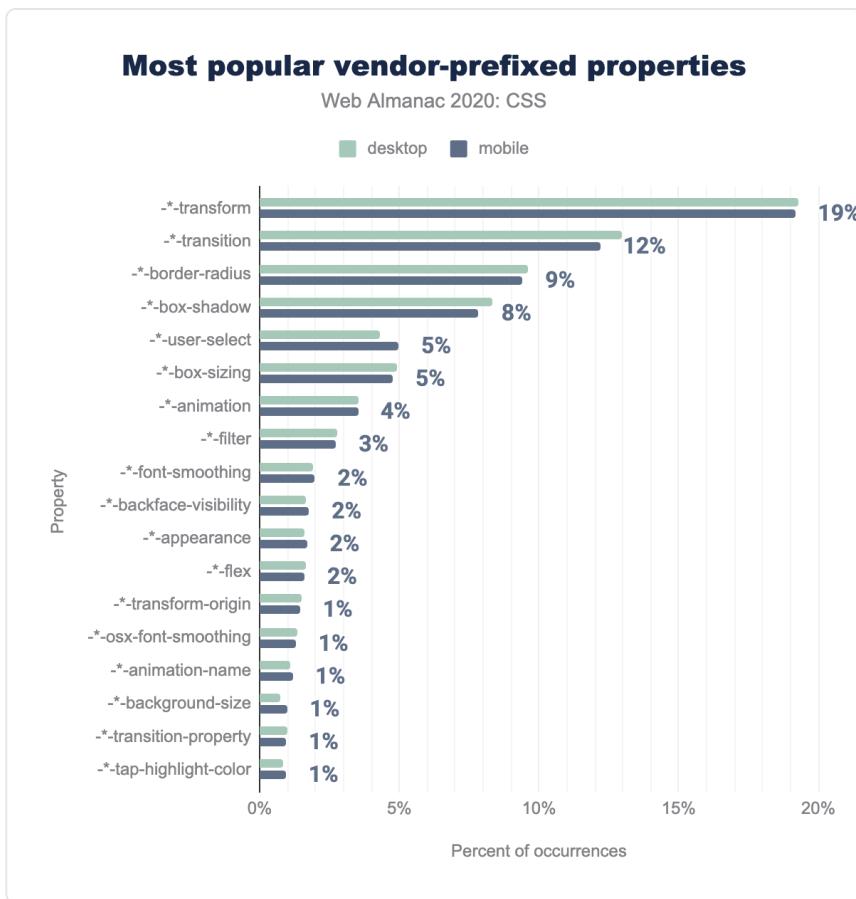


Figure 1.48. Relative popularity of properties that are most used with vendor prefixes, as a percent of occurrences.

## Most popular vendor prefixes

Web Almanac 2020: CSS

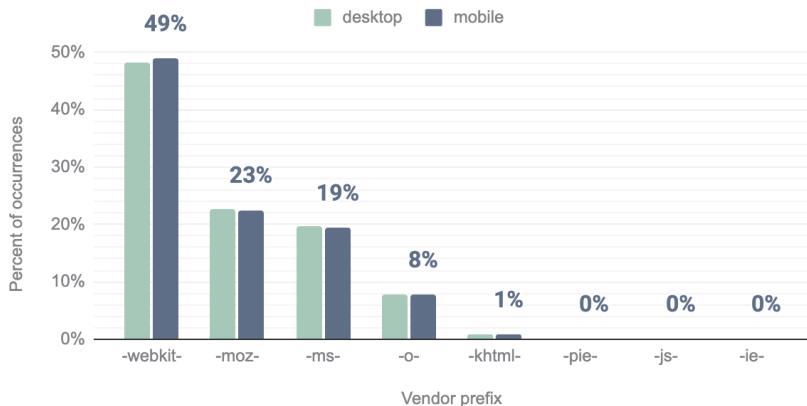


Figure 1.49. Relative popularity of vendor prefixes, as a percent of occurrences.

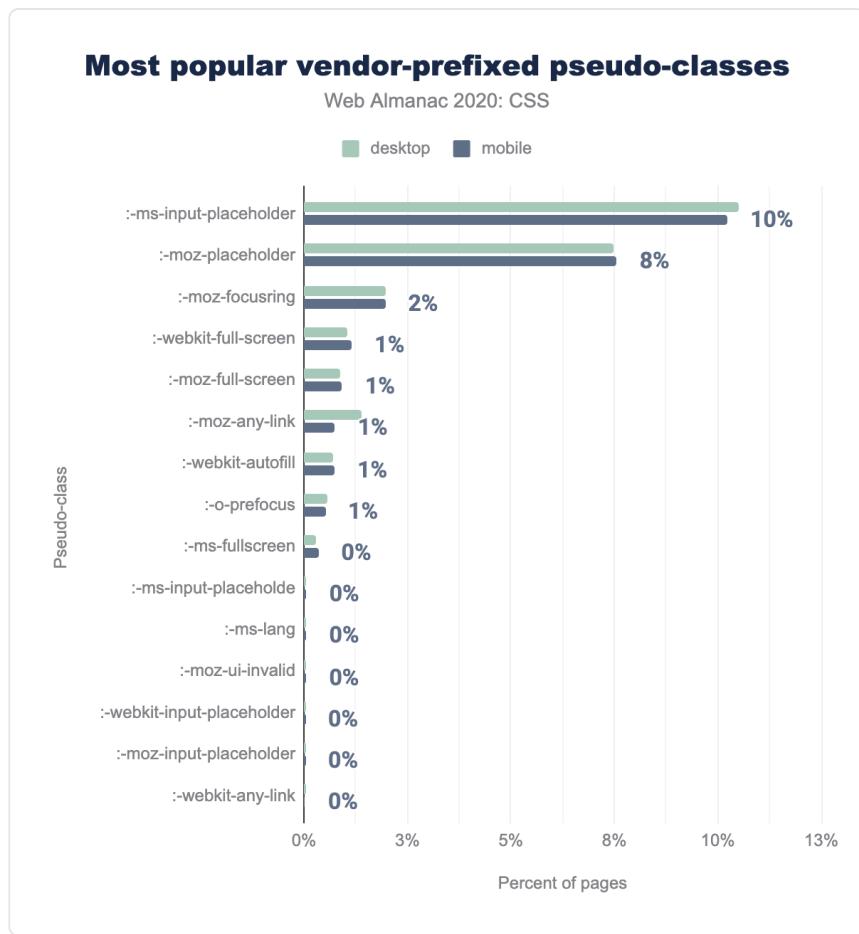


Figure 1.50. The most popular vendor-prefixed pseudo-classes as a percent of pages.

## Usage of prefixed pseudo-elements by category

Web Almanac 2020: CSS

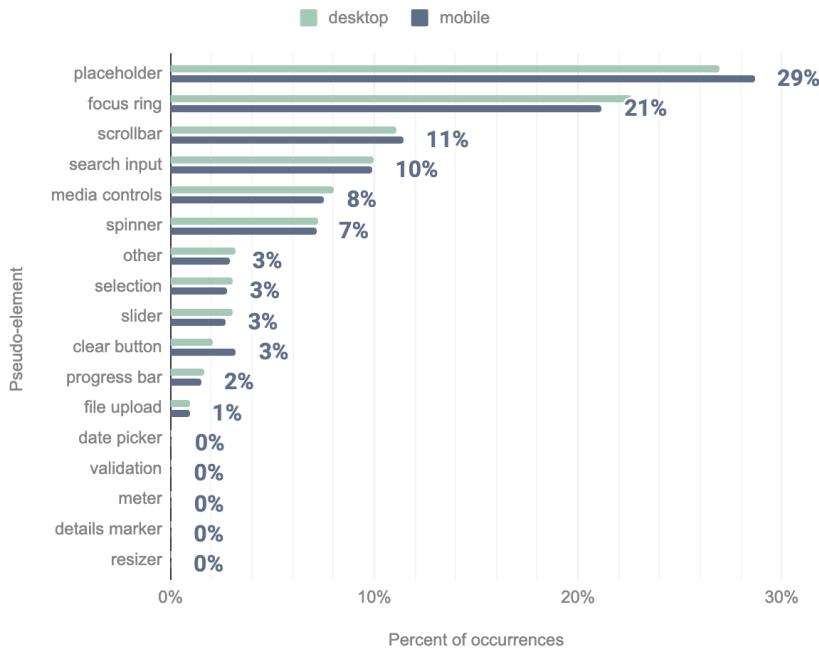


Figure 1.51. Relative popularity of vendor-prefixed pseudo-elements by purpose as a percent of occurrences.

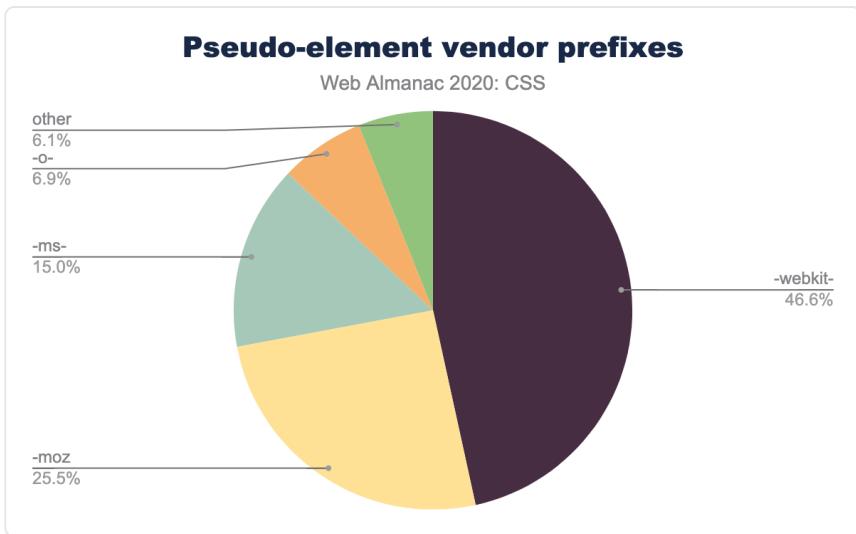


Figure 1.52. Relative popularity of pseudo-element vendor prefixes as a percent of occurrences on mobile pages.

# 98.22%

Figure 1.53. Percent of occurrences of vendor-prefixed functions on mobile pages that specify gradients.

## Vendor-prefixed media features

Web Almanac 2020: CSS

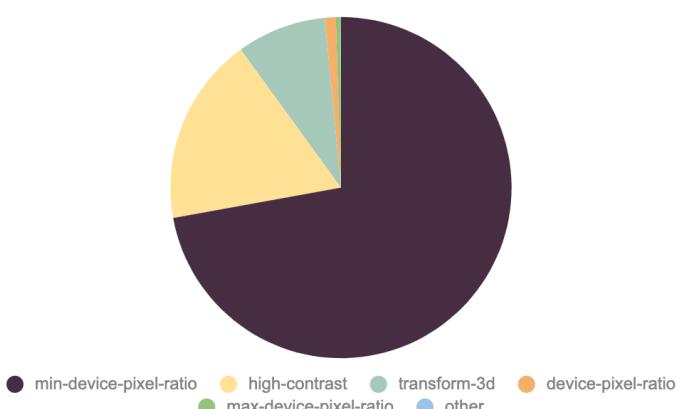


Figure 1.54. Relative popularity of vendor-prefixed media features as a percent of occurrences on mobile pages.

## Feature queries

### Most popular features queried

Web Almanac 2020: CSS

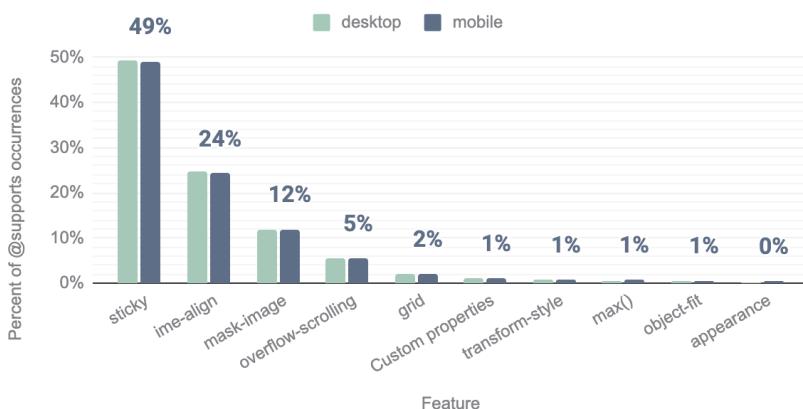


Figure 1.55. Relative popularity of `@supports` features queried as a percent of occurrences.

## Meta

### Declaration repetition

<i>Percentile</i>	<i>Unique / Total</i>
10	30.97%
50	45.43%
90	63.67%

Figure 1.56. Distribution of repetition ratios on mobile pages.

### Shorthands and longhands

#### Shorthands before longhands

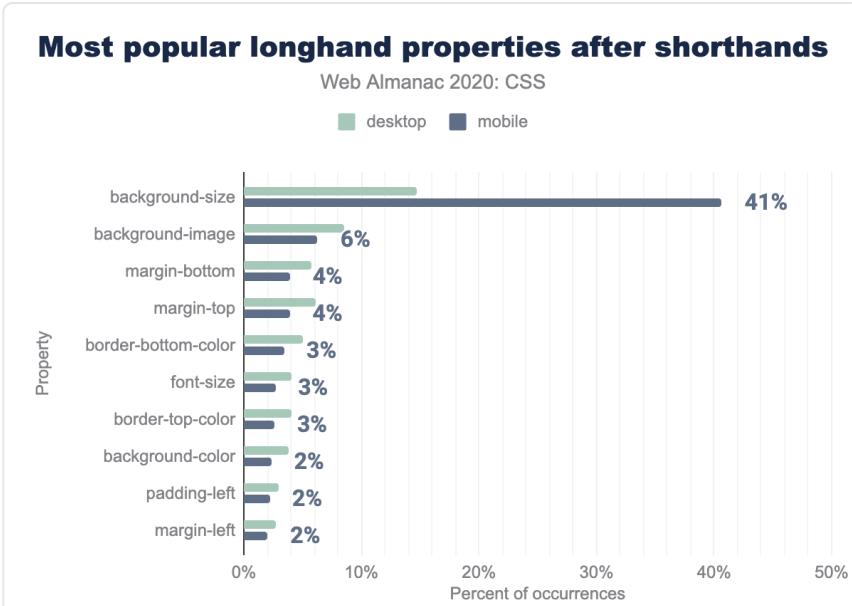


Figure 1.57. Most popular longhand properties after shorthands.

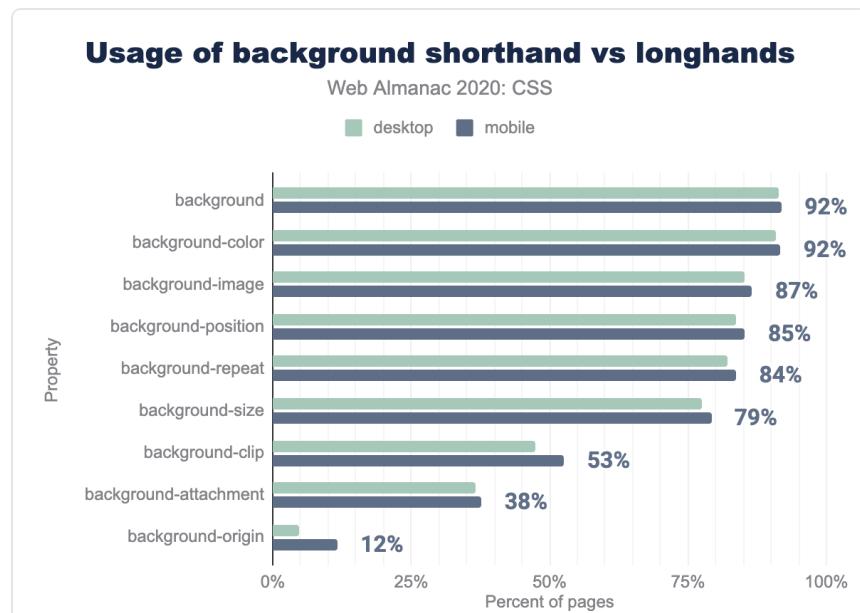


Figure 1.58. TODO.

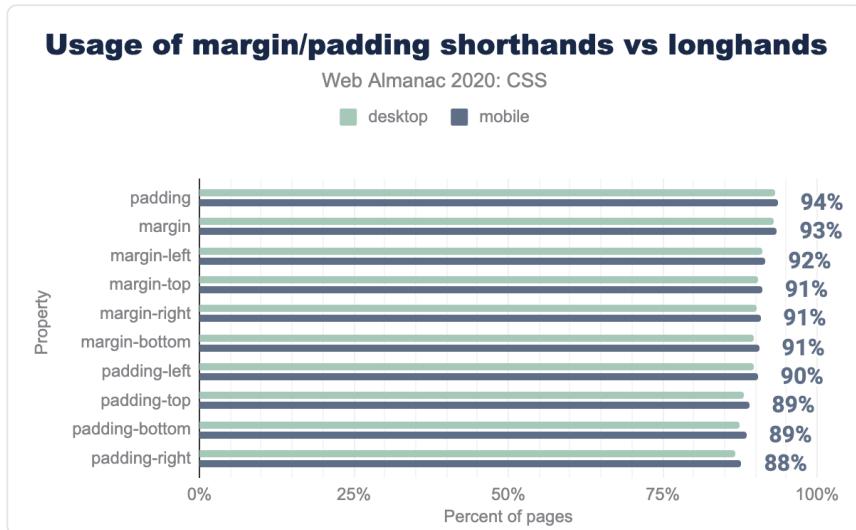
**font****background****Margins and paddings**

Figure 1.59. Usage of margin/padding shorthands vs longhands.

## Flex

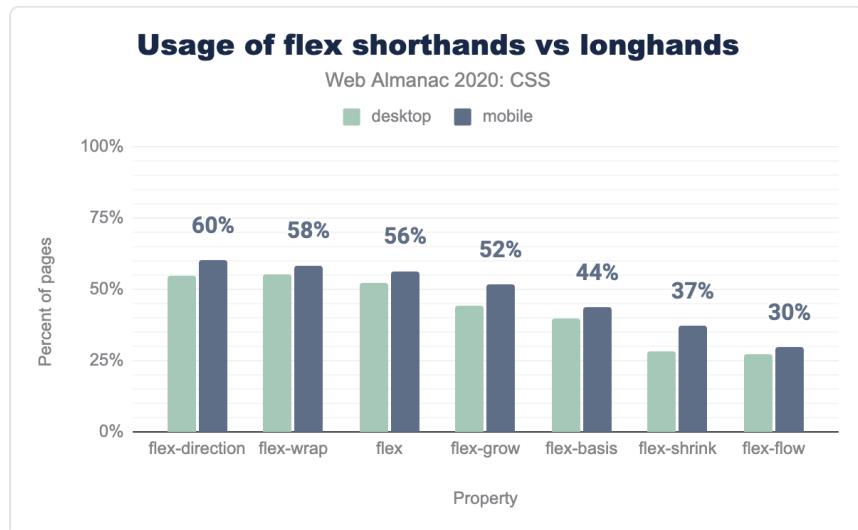


Figure 1.60. Usage of flex shorthands vs longhands.

## Grid

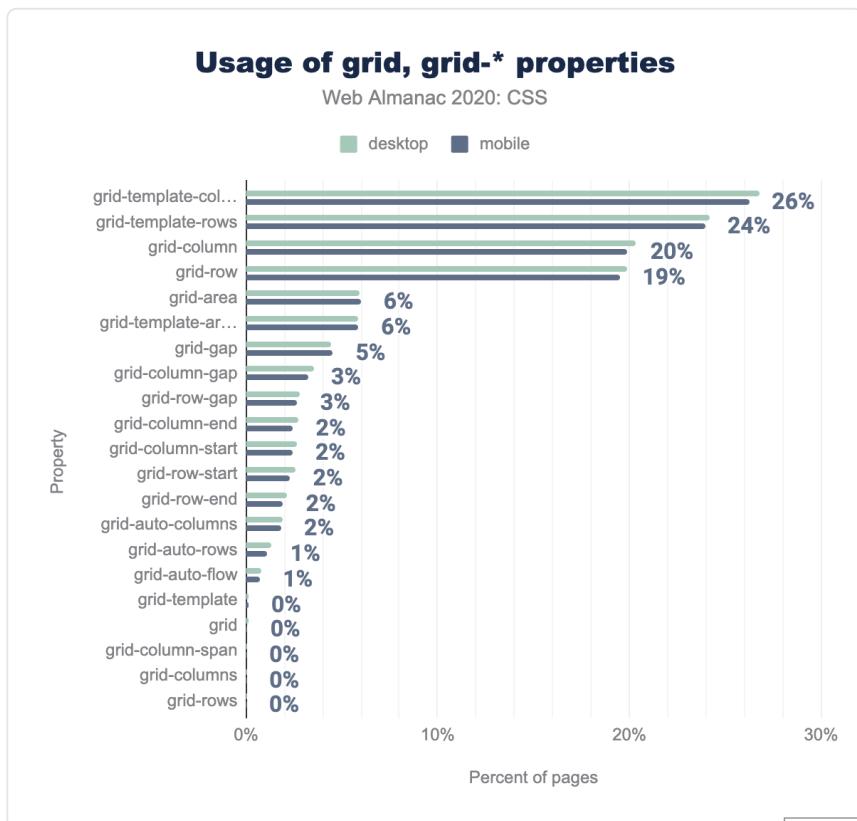


Figure 1.61. Usage of grid, grid-\* properties.

## CSS mistakes

### Syntax errors

#### Nonexistent properties

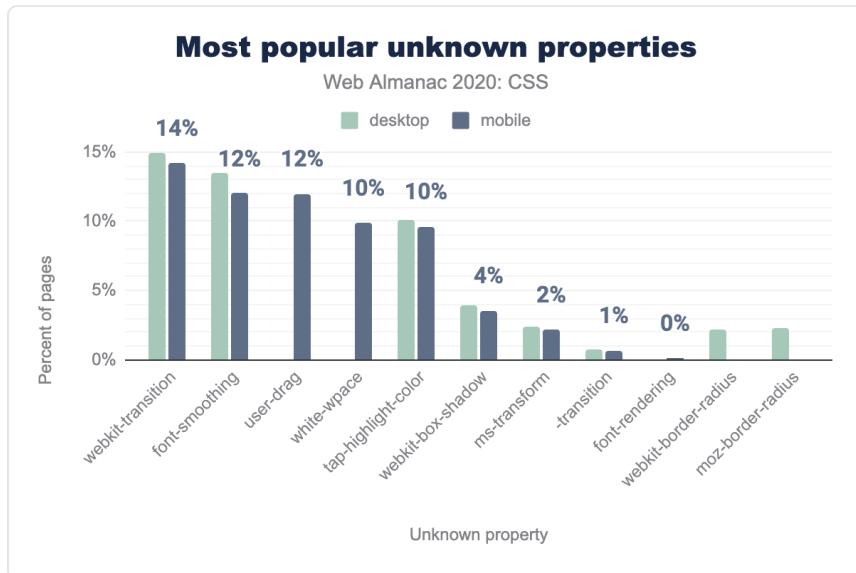


Figure 1.62. Most popular unknown properties.

## Longhands before shorthands

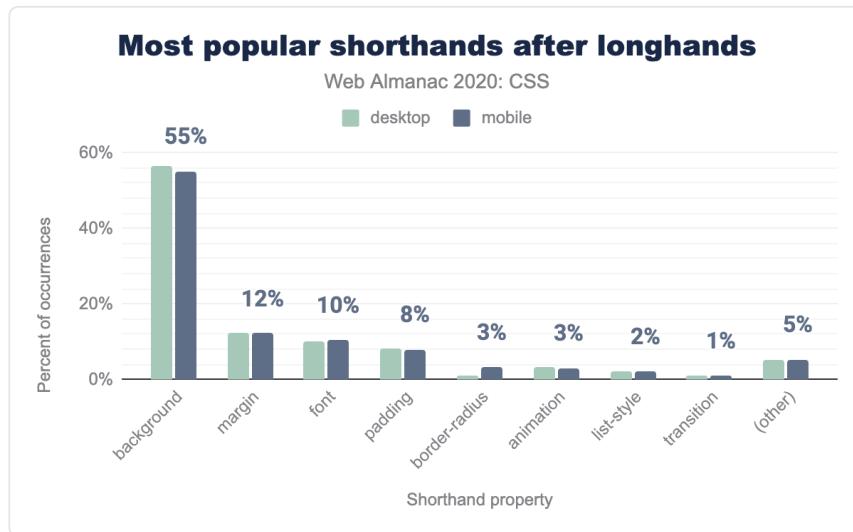


Figure 1.63. Most popular shorthands after longhands.

## Sass

### Most popular Sass function calls

Web Almanac 2020: CSS

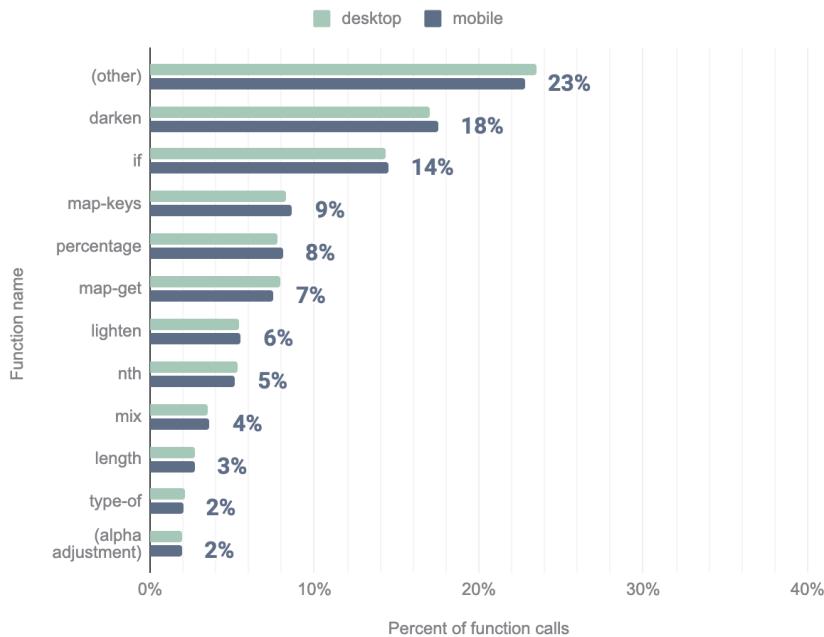


Figure 1.64. Most popular Sass function calls.

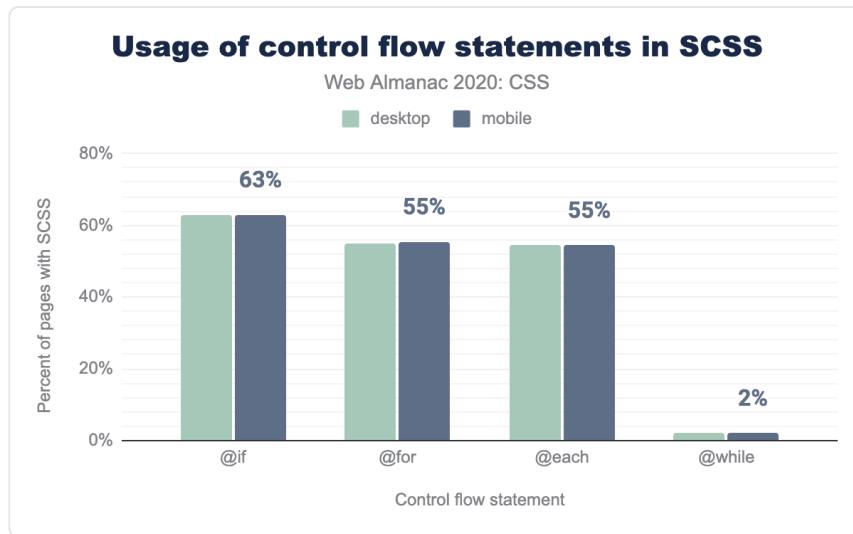


Figure 1.65. Usage of control flow statements in SCSS.

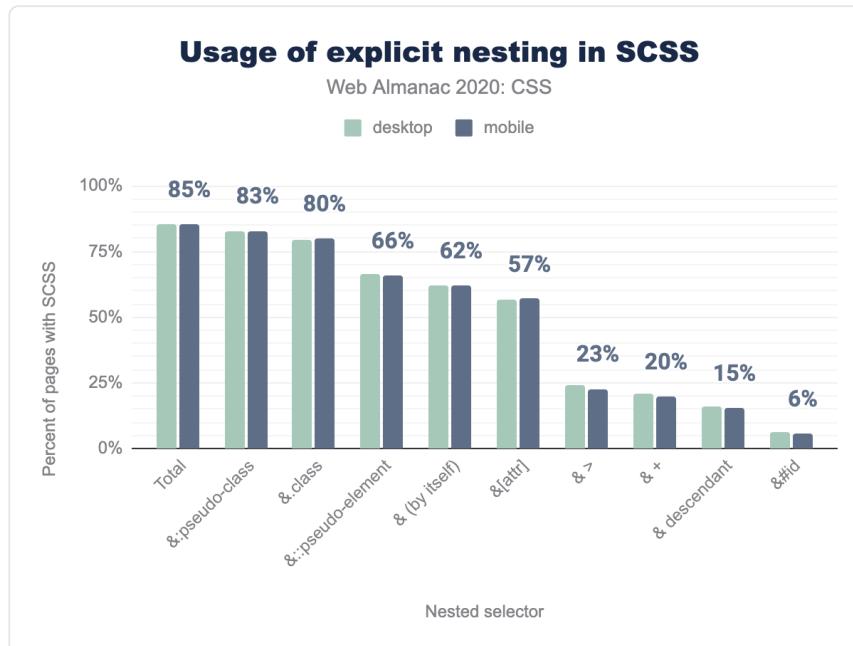


Figure 1.66. usage-of-explicit-nesting-in-scss.

# Conclusion

## Authors

---



Lea Verou

@leaverou LeaVerou <http://lea.verou.me/>



Chris Lilley

@svgeesus svgeesus <http://svgees.us/>



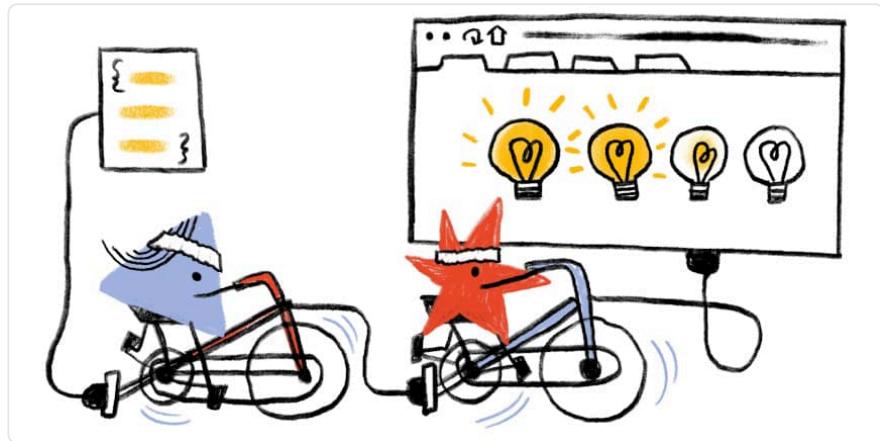
Rachel Andrew

@rachelandrew rachelandrew <https://rachelandrew.co.uk/>

---

## Part I Chapter 2

# JavaScript [UNEDITED]



Written by Tim Kadlec

Reviewed by Sawood Alam, Jad Joubran, Ahmad Awais, and Artem Denysov

Analyzed by Rick Viscomi and Paul Calvano

### Author



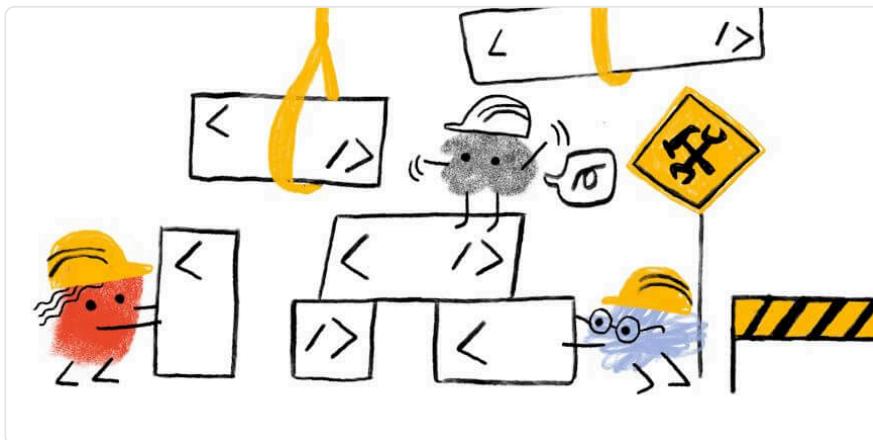
Tim Kadlec

[@tkadlec](#) [tkadlec](#)



# Part I Chapter 3

# Markup



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

Reviewed by Simon Pieters, Manuel Matuzovic, and Brian Kardell

Analyzed by Tony McCreadh

## Introduction

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

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

Last year's analysis was based on 5.8 million pages, of which 4.4 million were tested on desktop

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

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

## General

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

### Doctypes

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

Figure 3.1. Percent of pages with a doctype.

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

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

Figure 3.2. The 5 most popular doctypes.

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

Two things stand out from these results:

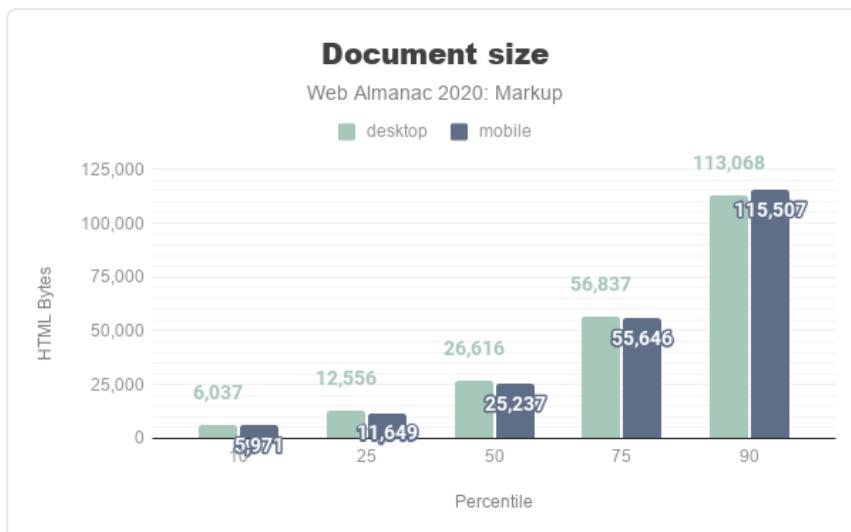
1. Almost 10 years after the announcement of living HTML (aka "HTML5"), living HTML has clearly become the norm.
2. The web before living HTML can still be seen in the next most popular doctypes, like XHTML 1.0. XHTML. Although their documents are likely delivered as HTML with a MIME type of `text/html`,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

## Document size

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

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

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



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

## Document language

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

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

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

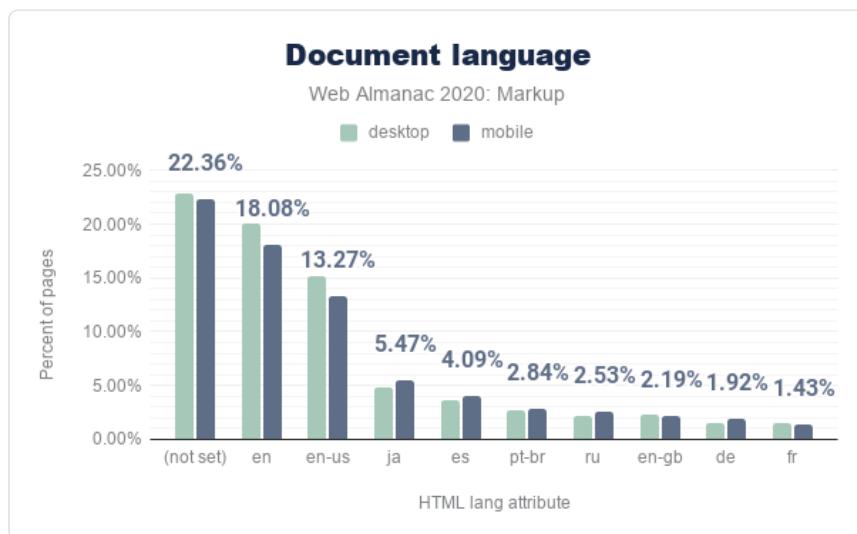


Figure 3.4. The top HTML `lang` attributes.

## Comments

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

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

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

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

## Conditional comments

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

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

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

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

## Script use

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

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

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

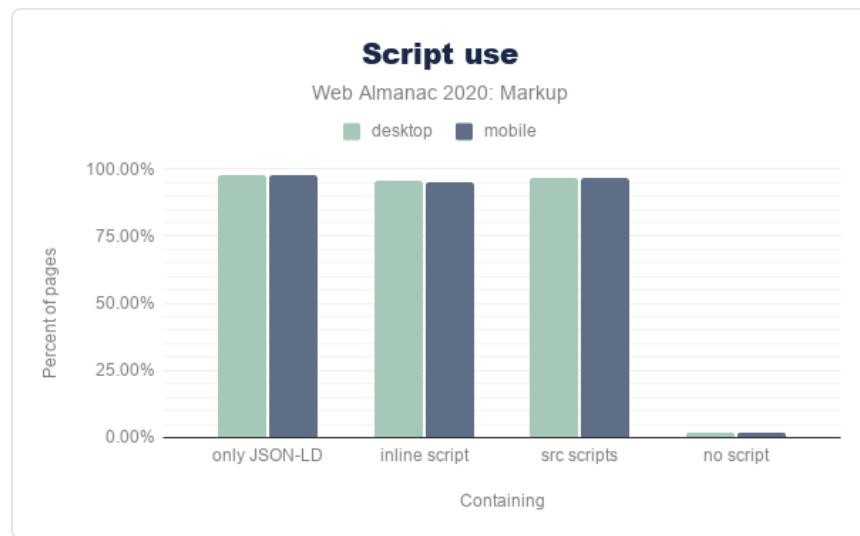


Figure 3.5. Usage of the `script` element.

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

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

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

## Script types

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

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

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

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

36.38% of all scripts have no values set whatsoever.

## Elements

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

### Element diversity

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

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

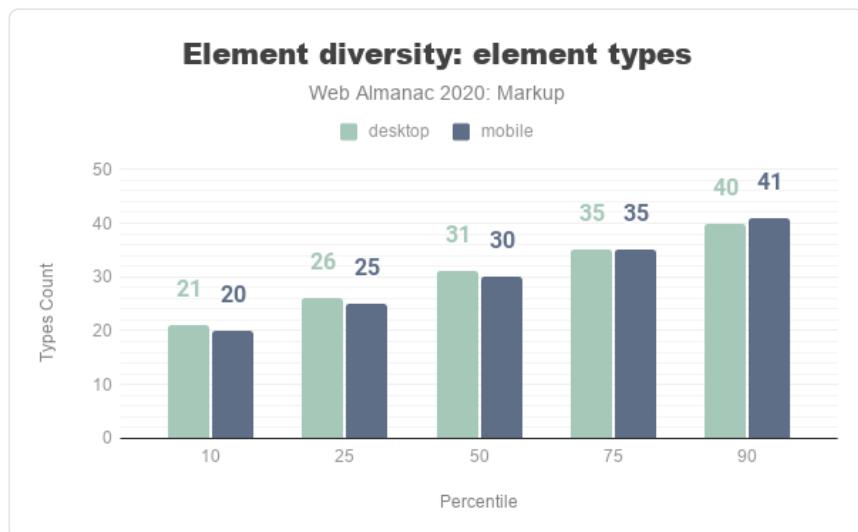


Figure 3.6. Distribution of the number of element types per page.

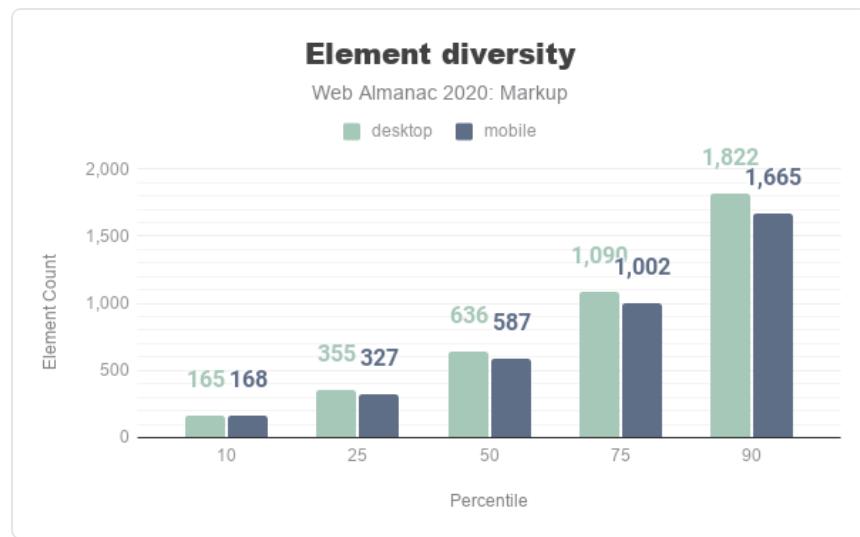


Figure 3.7. Distribution of the total number elements per page.

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

How are these elements distributed?

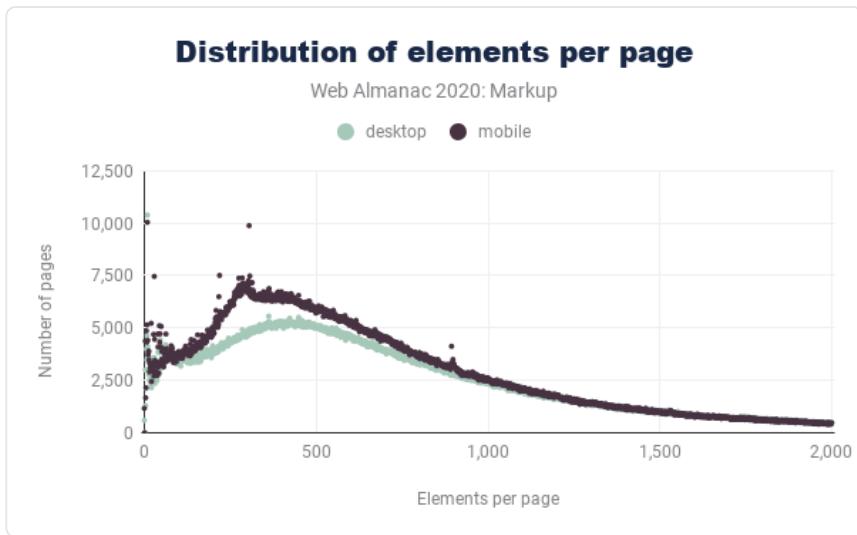


Figure 3.8. Distribution of the total number of elements per page.

Not that much changed compared to 2019!

## Top elements

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

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

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

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

### details and summary

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

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

```
<details>
<summary>Status: Operational</summary>
```

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

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

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

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

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

## Probability of element use

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

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

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

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

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

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

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

## Custom elements

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

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

Figure 3.13. The 14 most popular custom elements.

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

Other groups that stand out include AMP markup with `amp-` elements like `amp-img` (11,700 cases), `amp-analytics` (10,256) and `amp-auto-ads` (7,621), as well as Angular `app-` elements like `app-root` (16,314), `app-footer` (6,745), and `app-header` (5,274).

## Obsolete elements

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

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

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

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

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

### isindex

If you were wondering: The total number of `isindex` elements in this dataset is: one. Exactly one page used an `isindex` element. It was part of the specs until HTML 4.01 and XHTML 1.0, yet only properly specified in 2006 (aligning with how it was implemented in browsers), and then removed in 2016.

## Proprietary and made-up elements

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

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

Figure 3.15. Elements of questionable heritage.

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

## Headings

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

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

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

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

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

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

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

## Attributes

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

### Top attributes

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

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

Figure 3.18. Top 10 attributes by frequency of use.

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

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

## Attributes on pages

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

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

Figure 3.19. Top 10 attributes by page.

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

## **data-\* attributes**

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

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

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

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

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

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

## Miscellaneous

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

### viewport specifications

The `viewport` meta element is used to control layout on mobile browsers. While years ago, the

motto was kind of "don't forget the viewport meta element" when building a web page, eventually this became a common practice and the motto changed to "make sure zooming and scaling are not disabled."

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

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

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

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

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

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

## Favicons

The situation around favicons is fascinating. Favicons work with or without markup—some browsers would fall back to looking at the domain root—, accept several image formats, and then also promote several dozen sizes (some tools are reported to generate 45 of them;

[realfavicongenerator.net](#) would return 37 if requested to handle every case). As of this time of writing, there is an open issue for the HTML spec to help improve the situation.

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

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

Figure 3.22. Common favicon formats.

There are a couple of surprises in here:

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

## Button and input types

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

# 60.56%

Figure 3.23. Percent of pages with button elements.

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

Figure 3.24. Adoption of button types.

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

<b>Percentile</b>	<b>Buttons per page</b>
10	0
25	0
50	1
75	5
90	13

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

Pages in the 10th and 25th percentiles contain no buttons at all, while a page in the 90th

percentile contains 13 native `button` elements. In other words, 10% of pages contain 13 or more buttons.

## Link targets

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

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

Figure 3.26. Adoption of link target protocols.

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

## Links in new windows

**71.35%**

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

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

Elements	Pages
<code>&lt;a target="_blank" rel="noopener noreferrer"&gt;</code>	13.63%
<code>&lt;a target="_blank" rel="noopener"&gt;</code>	14.14%
<code>&lt;a target="_blank" rel="noreferrer"&gt;</code>	0.56%

Figure 3.28. Blank relationships.

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

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

## Conclusion

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

**3.97%**

Figure 3.29. Percent of pages with a quirky doctype.

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

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

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

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

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

And still, there are no signs we exhausted the options HTML gives us. The median of 30 different elements used on a given page, which is roughly a quarter of the elements HTML provides us with, suggests a rather one-sided use of HTML. That is supported by the immense popularity of elements like `div` and `span`, and no custom elements to potentially meet the demands that these two elements may represent. Unfortunately, we couldn't validate each document in the sample; however, anecdotally and to be taken with caution, we learned that 79% of W3C-tested documents have validation errors. After everything we've seen, it looks like we're still far from mastering the craft of HTML.

That compels us to close with an appeal: Pay attention to HTML. Focus on HTML. It's important and worthwhile to invest in HTML. HTML is a document language that may not have the charm of a programming language, and yet the web is built on it. Use less HTML and learn what's really needed. Use more appropriate HTML—learn what's available and what it's there for. And

validate your HTML. Anyone can write invalid HTML (just invite the next person you meet to write an HTML document and validate the output) but a professional developer can be expected to produce valid HTML. Writing correct and valid HTML is a craft to take pride in.

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

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

## Authors

---



### Jens Oliver Meiert

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

Jens Oliver Meiert is a web developer and author (*CSS Optimization Basics*<sup>1</sup>, *The Web Development Glossary*<sup>2</sup>), who works as an engineering manager at Jimdo<sup>3</sup>. He's an expert on web development where he specializes in HTML and CSS optimization. Jens contributes to technical standards and regularly writes about his work and research, particularly on his website, [meiert.com](https://meiert.com)<sup>4</sup>.



### Catalin Rosu

@catalinred catalinred <https://catalin.red/>

Catalin Rosu is a front-end developer at Caphyon<sup>5</sup> and currently works on Wattspeed<sup>6</sup>. He has a passion for web standards and a keen eye for great UX & UI, things he tweets<sup>7</sup> and writes about on his website<sup>8</sup>.



### Ian Devlin

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

Ian Devlin is a web developer who advocates for good, semantic HTML, as well as accessibility. He once wrote a book about *HTML5 Multimedia*<sup>9</sup>, and sporadically writes on his website<sup>10</sup> about the Web and other things. He currently works as a Senior Frontend Engineer at real.digital<sup>11</sup> in Germany.

---

1. <https://leanpub.com/css-optimization-basics>

2. <https://leanpub.com/web-development-glossary>

3. <https://www.jimdo.com/>

4. <https://meiert.com/en/>

5. <https://www.caphyon.com/>

6. <https://www.wattspeed.com/>

7. <https://twitter.com/catalinred>

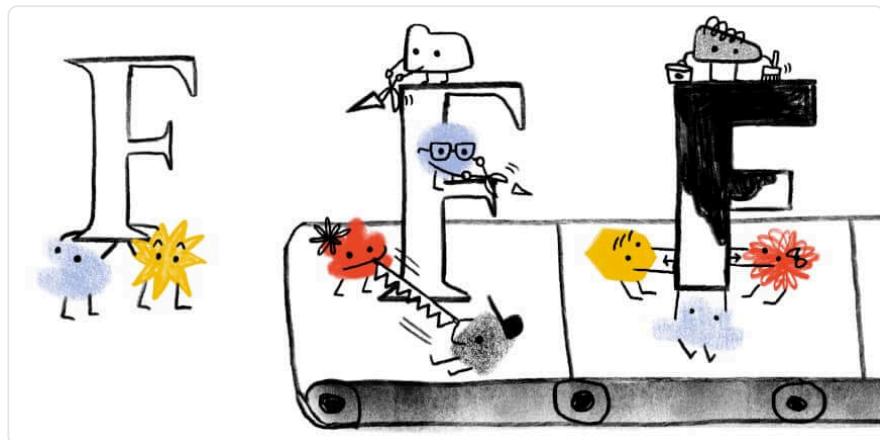
8. <https://catalin.red/>

- 
9. <https://www.peachpit.com/store/html5-multimedia-develop-and-design-9780321793935>
  10. <https://iandevlin.com/>
  11. <https://www.real-digital.de/>



# Part I Chapter 4

# Fonts [UNEDITED]



**Written by Raph Levien and Jason Pamental**

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

Analyzed by Abby Tsai

## Introduction

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

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

## Where are web fonts being used?

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

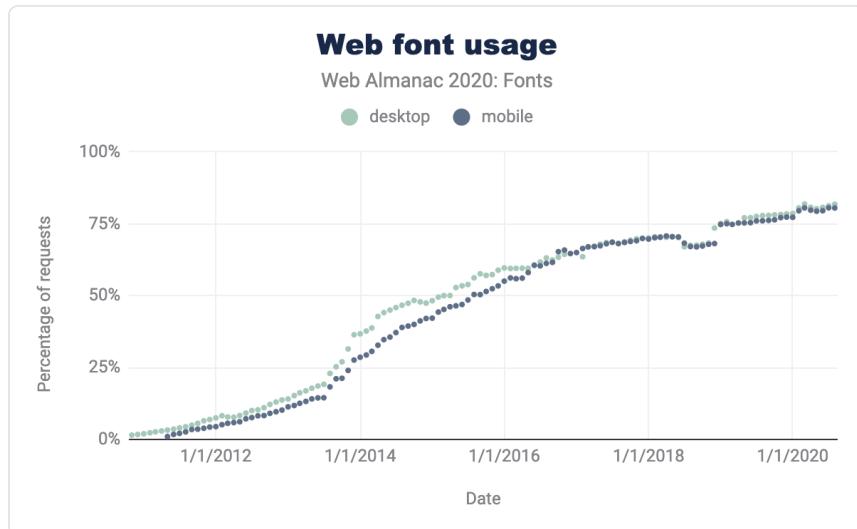


Figure 4.1. Web font usage over time.

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

<https://discuss.httparchive.org/t/how-does-web-font-usage-vary-by-country/1649>

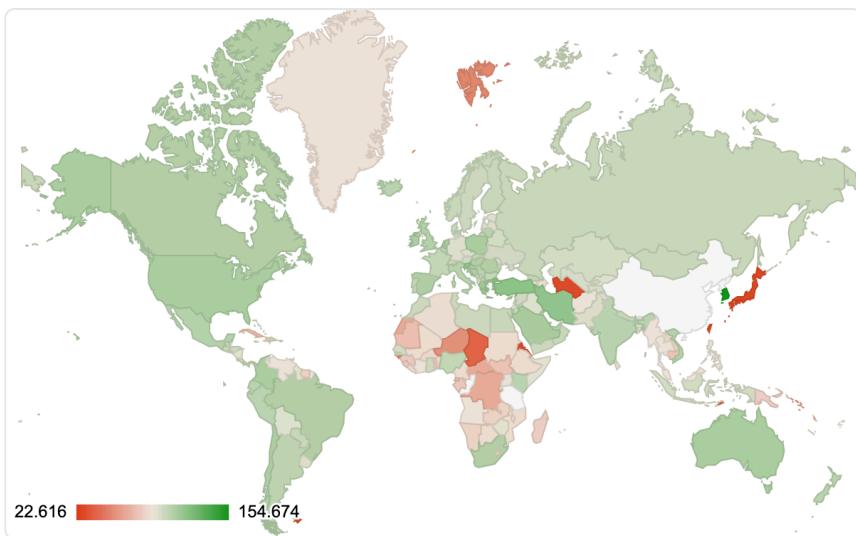


Figure 4.2. Web fonts usage by country (desktop).

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

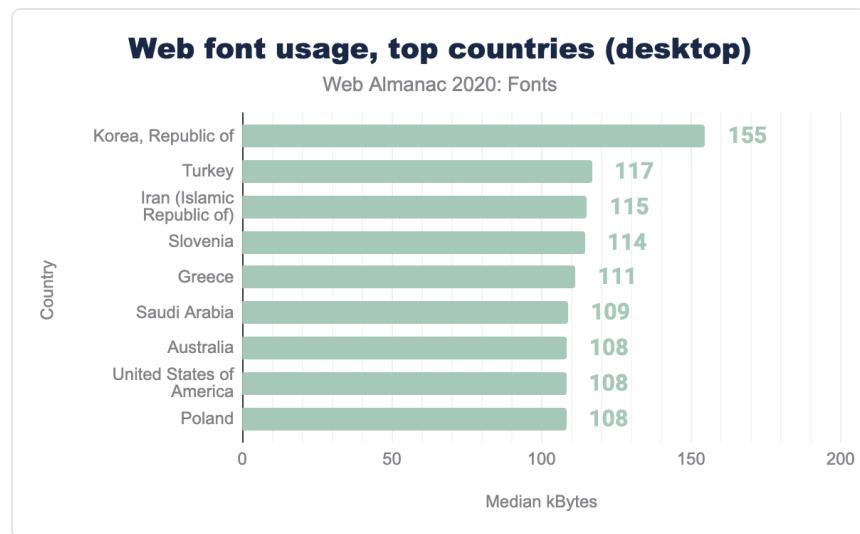


Figure 4.3. Web fonts usage, top countries (desktop).

There's a great thread on web font usage by country on the [httparchive](#) discussion forum.

## Serving with a service

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

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

## Not all services have the same service

It was interesting to note the differences in speed from the various free/open source and commercial services. When looking at First Content Paint (FCP) and Last Content Paint (LCP)

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

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

### Local isn't always better

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

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

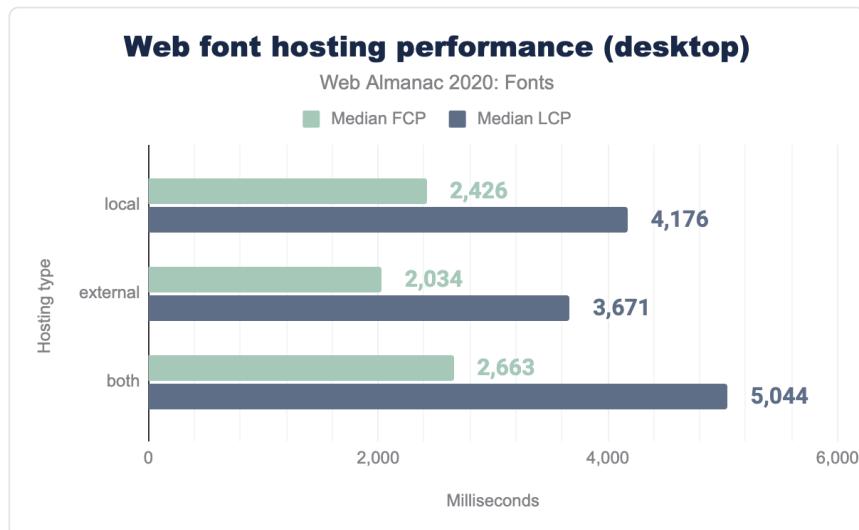


Figure 4.4. Web font hosting performance, desktop.

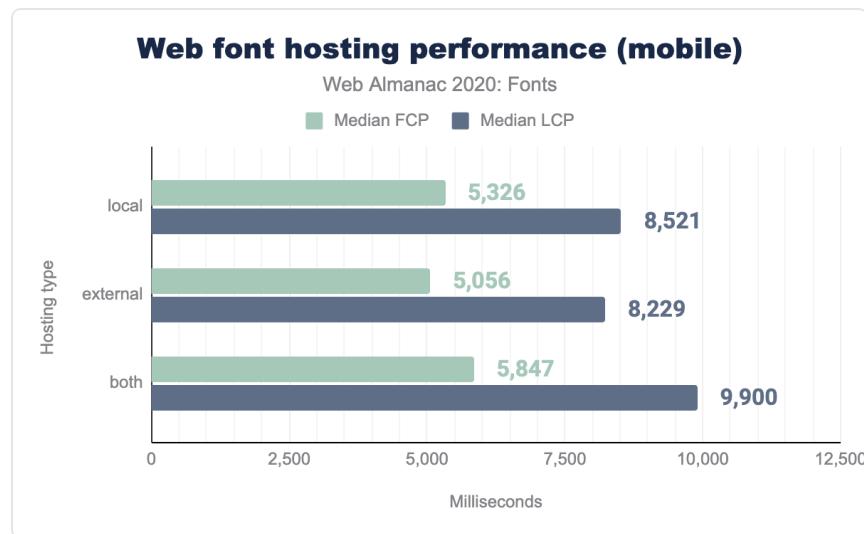


Figure 4.5. Web font hosting performance, mobile.

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

## Racing to first paint

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

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

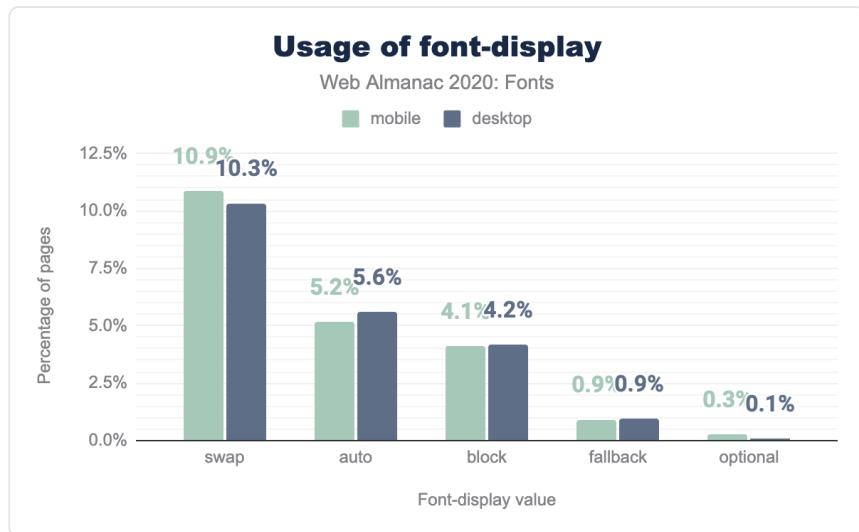


Figure 4.6. Usage of font-display.

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

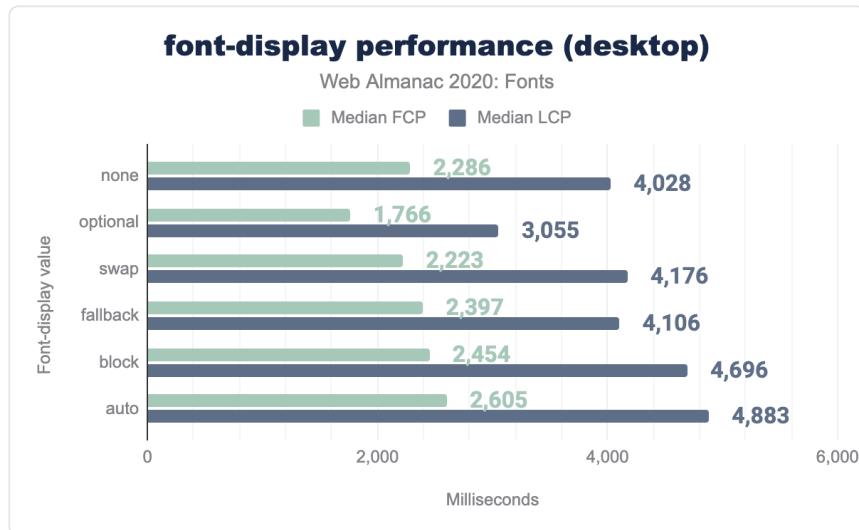


Figure 4.7. font-display, performance, desktop.

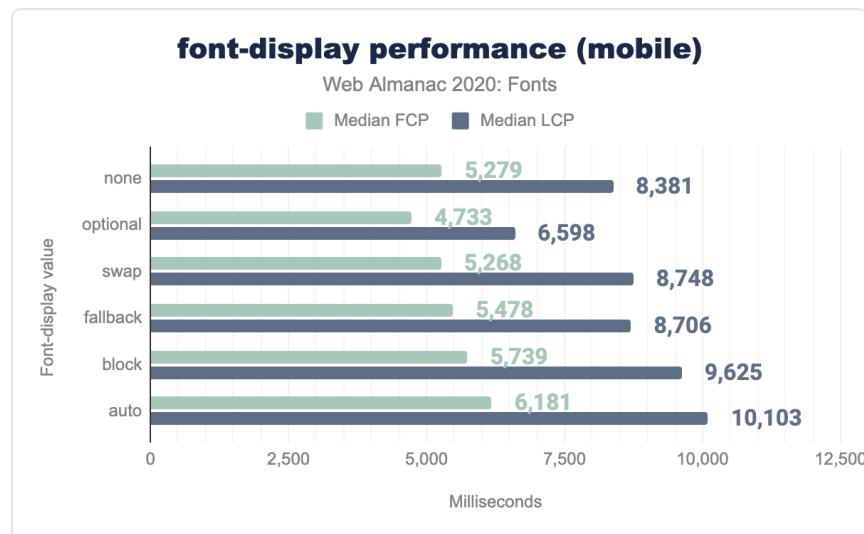


Figure 4.8. *font-display*, performance, mobile.

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

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

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

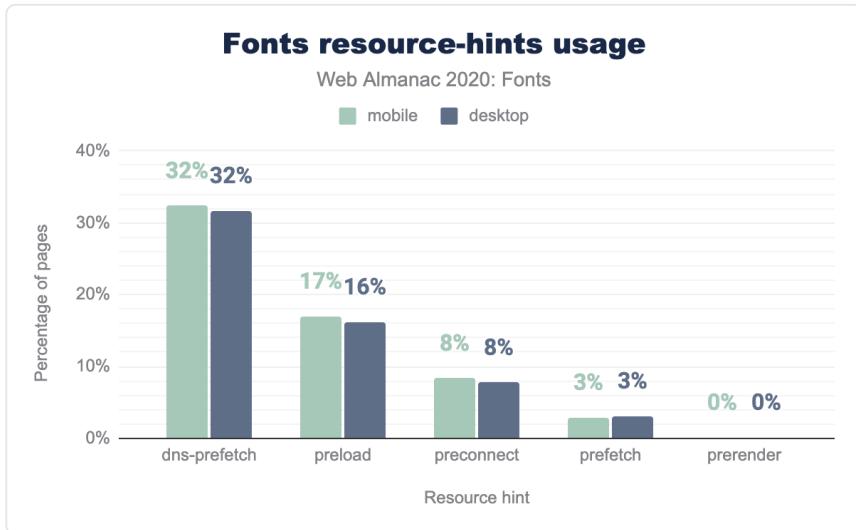
## Resource hints

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

Ordinarily, fetching web fonts is a two-stage process. The first stage is loading the CSS, which contains a reference (in `@font-face` sections) to the actual font binaries. Only then can the

connection to that server begin, which further breaks down into the DNS query for the server, and actually initiating a connection (which, these days, usually involves an HTTPS cryptographic handshake).

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



*Figure 4.9. resource-hints use on fonts.*

## Font resource-hint performance (desktop)

Web Almanac 2020: Fonts

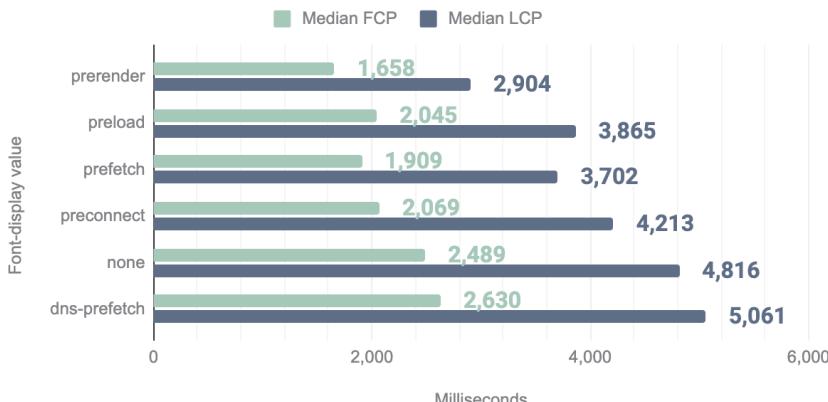


Figure 4.10. resource-hints performance, desktop.

## Font resource-hint performance (mobile)

Web Almanac 2020: Fonts

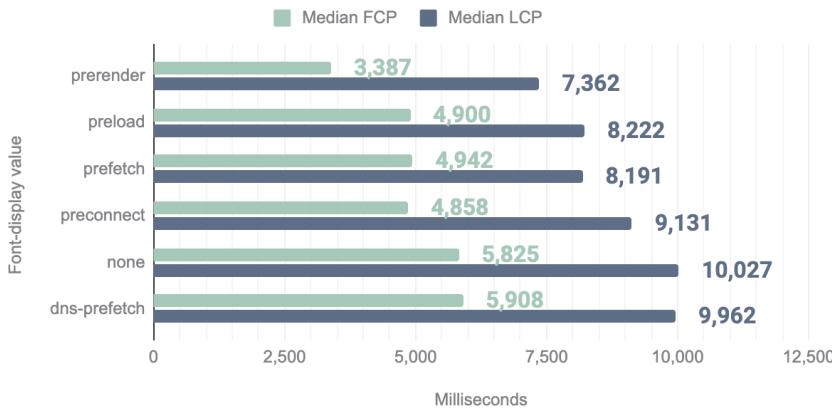


Figure 4.11. resource-hints performance, mobile.

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

before the CSS link:

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

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

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

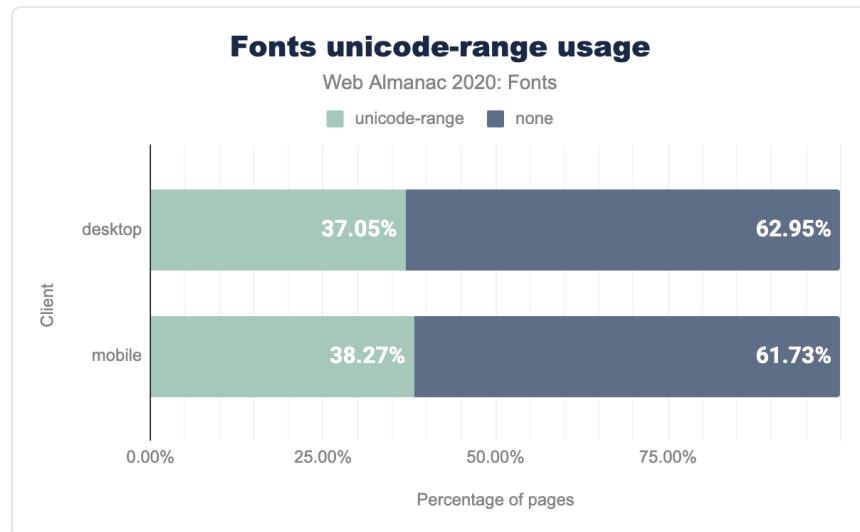
## Home on the (Unicode) range

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

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

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

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

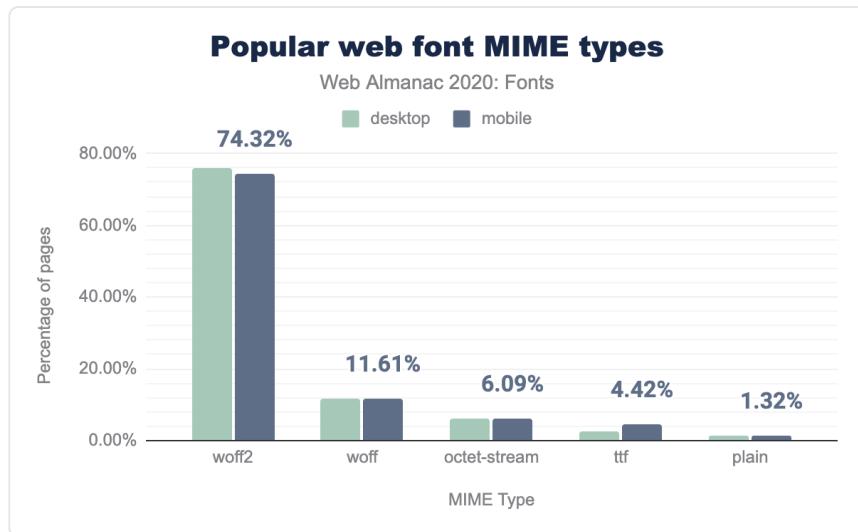


*Figure 4.12. usage of `unicode-range`.*

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

## Formats and MIME types

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



*Figure 4.13. Popular web font MIME types.*

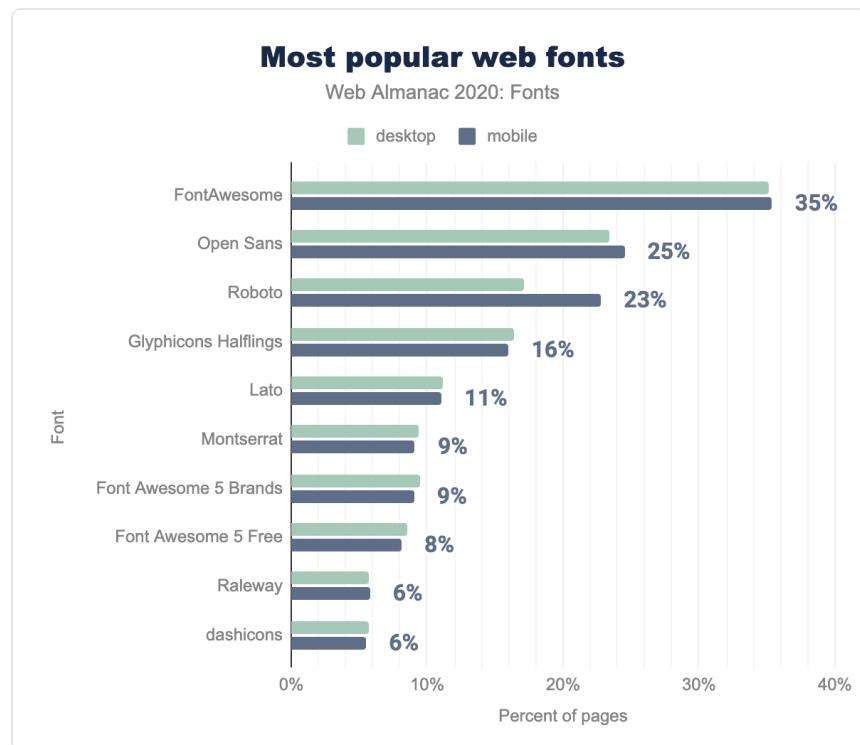
WOFF is an older, less efficient compression mechanism, but almost universally supported, accounting for an additional 11.6% of fonts served. In almost all cases (Internet Explorer 9-11 being the main exception), serving a font as WOFF is leaving performance on the table, and shows a risk of self-hosting;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

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

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

## Popular fonts

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



*Figure 4.14. Popular typefaces.*

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

## Color fonts

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

The COLR vector format is supported on all major modern browsers, but only fairly recently. The fourth format is embedding SVG in OpenType (not to be confused with SVG fonts), but not

supported in Chrome. One drawback of SVG in OpenType is lack of support for font variations, an increasingly important aspect of modern Web design. For this reason, the COLR format is likely to prevail, particularly as support for gradients and clipping is being developed for a future version of COLR. Vector formats are usually much smaller than images, and also scale cleanly to larger sizes, so when COLR arrives with a richer shading model, it could well become popular.

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

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

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

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

Lots more information about color fonts, including examples, are available at [colorfonts.wtf](http://colorfonts.wtf).

## Variable fonts

### Usage of font-variation-settings axes

Web Almanac 2020: Fonts

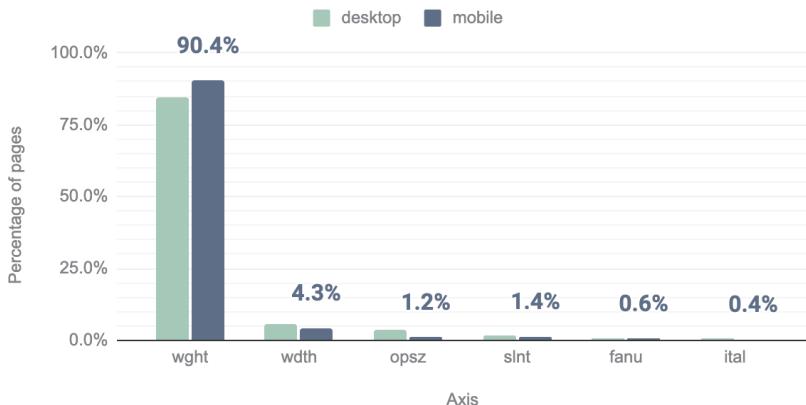


Figure 4.15. Usage of font-variation-settings axes.

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

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

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

The optical size (`opsz`) feature is used for approximately 2% of the variable font usage. This is one to watch, as tuning the appearance of a font to match its intended size of presentation improves the visual refinement in perhaps subtle but very real ways. Usage is also likely to increase once some current cross-browser and cross-platform uncertainties on how the optical

sizes are defined are cleared up. One appealing aspect of the optical size feature is that with the `auto` setting, the variation happens automatically, so the developer gets the benefit of that refinement just by using a font with the `opsz` feature.

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

## Towards the future

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

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

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

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

There are various details to work out, including ones that will help ensure privacy and backwards compatibility—but initial research has been extremely promising and it's hoped this technology will reach the wider web sometime in the next couple years. You can learn more about it in this introduction by Jason Pamental, and read the full [Working Group Evaluation](#)

Report on the W3C site.

## Authors

---



### Raph Levien

🐦 @raphlinus Ⓜ raphlinus ✉ https://levien.com

Raph Levien has been working with fonts for over 35 years, including a PhD from UC Berkeley in font design tools. He is rejoining Google Fonts<sup>12</sup> as a font technology researcher, after having co-founded the team in 2010.



### Jason Pamental

🐦 @jpamental Ⓜ jpamental ✉ http://rwt.io

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

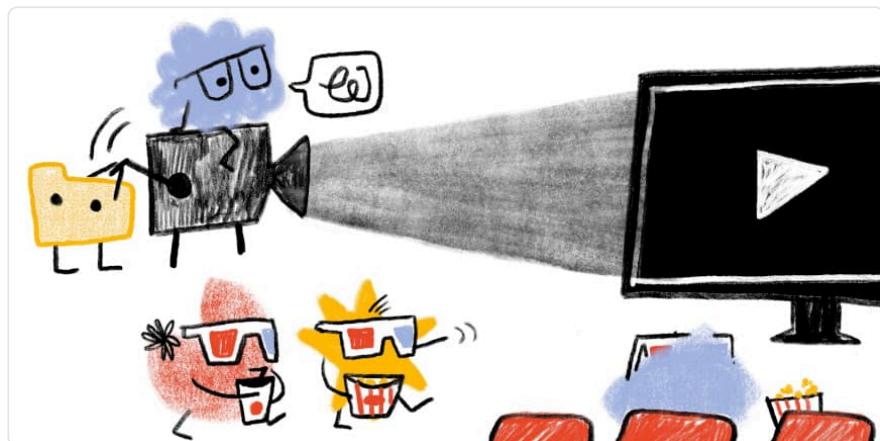
---

---

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

# Part I Chapter 5

# Media [UNEDITED]



*Written by Tamas Piros and Ben Seymour*

*Reviewed by Nicolas Hoizey, Simon Pieters, Colin Bendell, and Doug Sillars*

*Analyzed by Stefan Matei*

## Authors



Tamas Piros

✉ @tpiros    🌐 tpiros    🌐 <https://www.fullstacktraining.com>



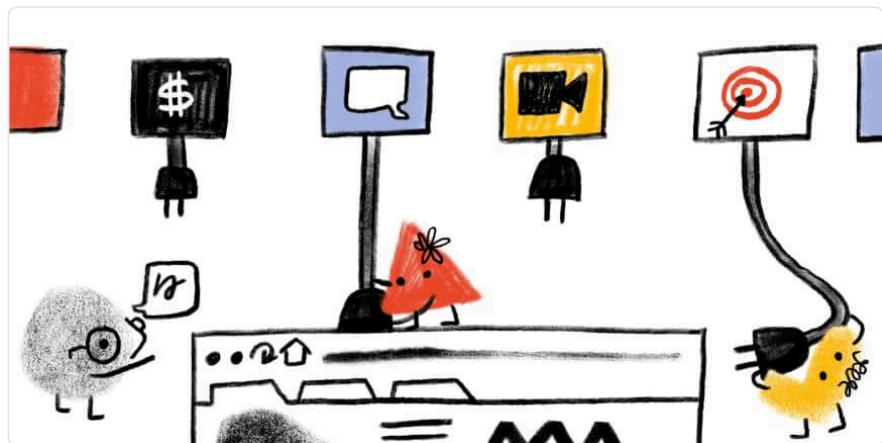
Ben Seymour

✉ bseymour    🌐 <http://benseymour.com>



# Part I Chapter 6

# Third Parties [UNEDITED]



Written by Simon Hearne

Reviewed by Tammy Everts and jzyang

Analyzed by Max Ostapenko

## Author



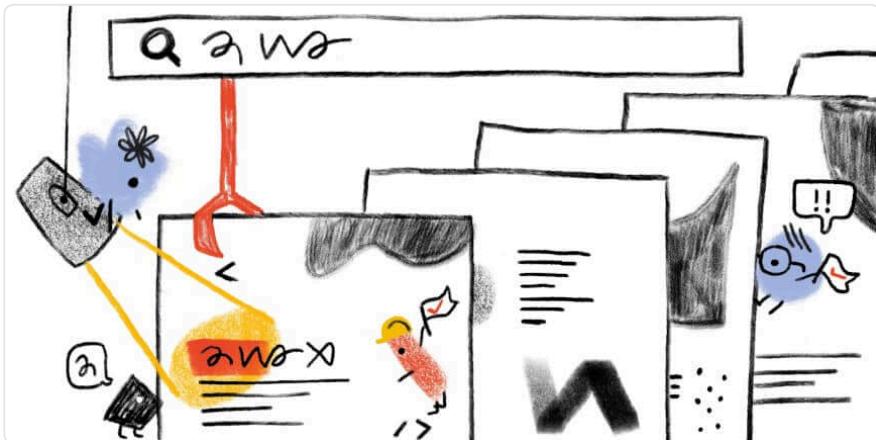
Simon Hearne

✉ @simonhearne Ⓛ simonhearne Ⓡ <https://simonhearne.com>



# Part II Chapter 7

# SEO



*Written by Aleyda Solis, Michael King, and Jamie Indigo*

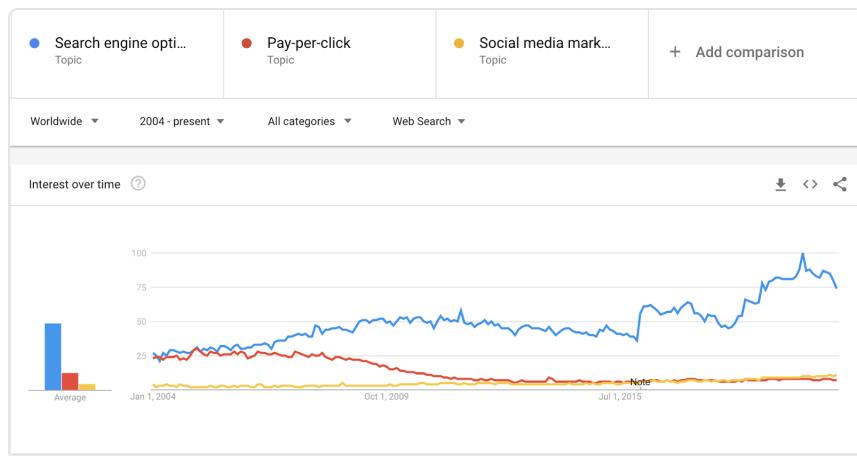
*Reviewed by Nate Dame, Catalin Rosu, Dave Sottimano, Dave Smart, Dustin Montgomery, Sawood Alam, and Barry Pollard*

*Analyzed by Tony McCreath and Antoine Eripert*

## Introduction

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

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



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

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

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

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

## Fundamentals

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

### Crawlability and indexability

Search engines use web crawlers (also called spiders) to discover new or updated content from websites, browsing the web by following links between pages. Crawling is the process of

---

looking for new or updated web content (whether web pages, images, videos, etc.).

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

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

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

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

### **robots.txt**

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

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

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

Overall, 80.46% of mobile pages responded to `robots.txt` with a 2XX response. Of these, 25.09% were not recognized as valid. This has slightly improved over 2019, when it was found that 27.84% of mobile sites had a valid `robots.txt`.

Lighthouse, the data source for testing `robots.txt` validity, introduced a `robots.txt` audit as

part of the v6 update. This inclusion highlights that a successfully resolved request does not mean that the cornerstone file will be able to provide the necessary directives to web crawlers.

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

Figure 7.2. `robots.txt` response codes.

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

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

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

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

Figure 7.3. `robots.txt` User-agent directives.

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

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

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

## Meta robots

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

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

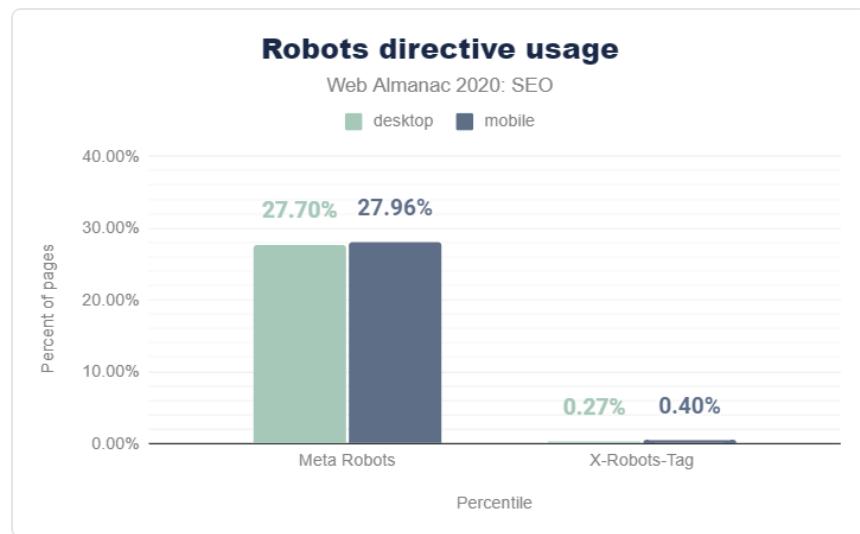


Figure 7.4. Usage of meta robots and `X-Robots-Tag` directives.

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

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

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

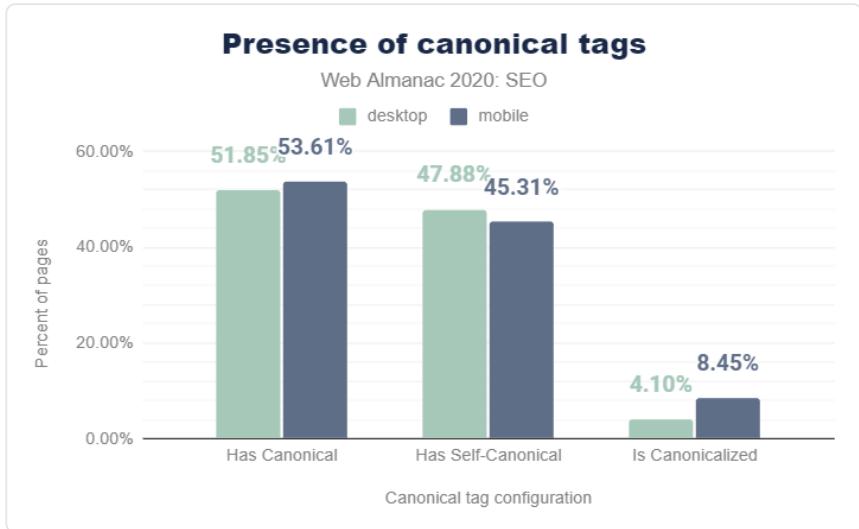
## Canonicalization

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

- The canonical tag configuration is used along with other signals to select the canonical URL of a page; it is not the only one.

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

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



*Figure 7.5. Usage of canonical tags.*

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

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

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

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

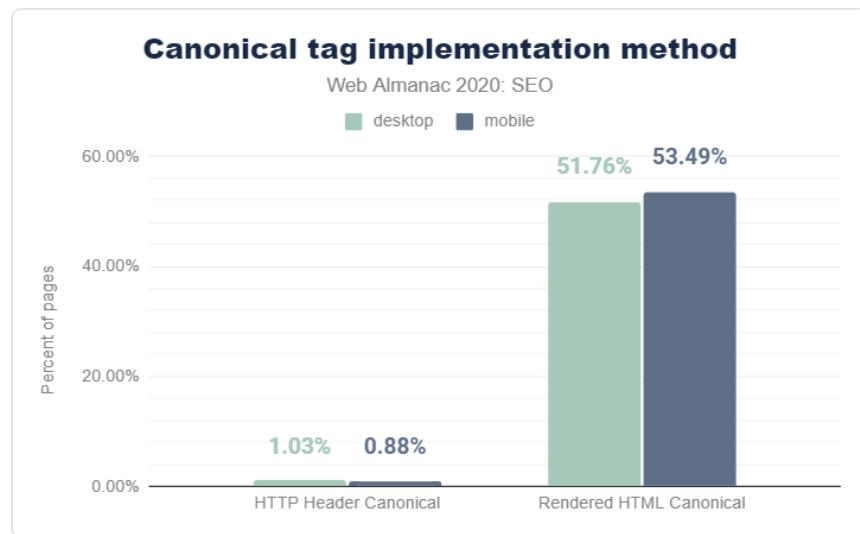


Figure 7.6. Usage of `HTTP header` and `HTML head` canonicalization methods.

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

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

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

## Content

The primary purpose that both search engines and Search Engine Optimization serve is to give visibility to content that users need. Search engines extract features from pages to determine what the content is about. In that way, the two are symbiotic. The features extracted align with signals that indicate relevance and inform ranking.

To understand what search engines are able to effectively extract, we have broken out the

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

## Rendered versus non-rendered text content

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

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

We assessed word count as a heuristic of that disparity.

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

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

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

We found that the median desktop site features 11.67% more words when rendered than it does on an initial crawl of its raw HTML. We also found that the median mobile site displays 13.33% less text content than its desktop counterpart. The median mobile site also displays 11.54% more words when rendered than its raw HTML counterpart.

Across our sample set, there are disparities across the combination of mobile/desktop and rendered/non-rendered. This suggests that although search engines are continually improving

in this area, most sites across the web are missing out on opportunities to improve their organic search visibility through a stronger focus on ensuring their content is available and indexable. This is also a concern because the lion's share of available SEO tools do not crawl in the above combination of contexts and automatically identify this as an issue.

## Headings

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

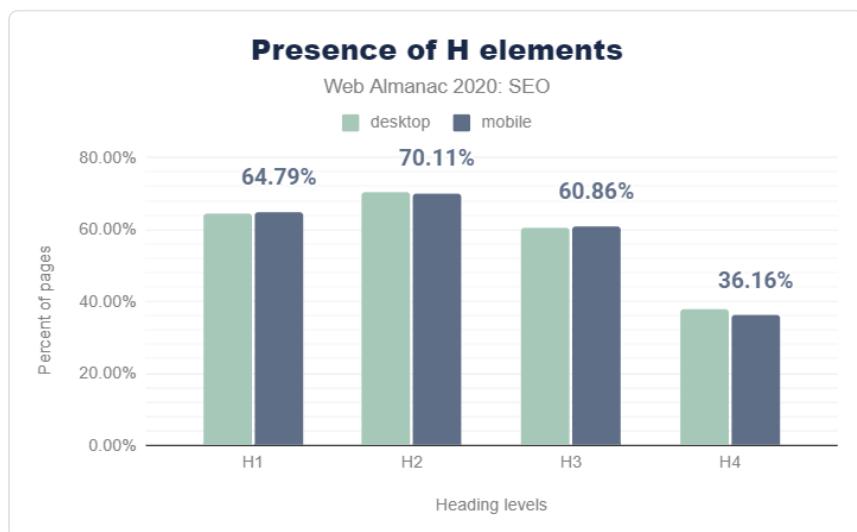


Figure 7.8. Usage of heading levels 1 through 4, including empty headings.

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

These numbers hover around 60%+ through `H2` and `H3`. The incidence rate of `H4` elements is lower than 4%, suggesting that the level of specificity is not required for most pages or the developers style other headings elements differently to support the visual structure of the content.

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

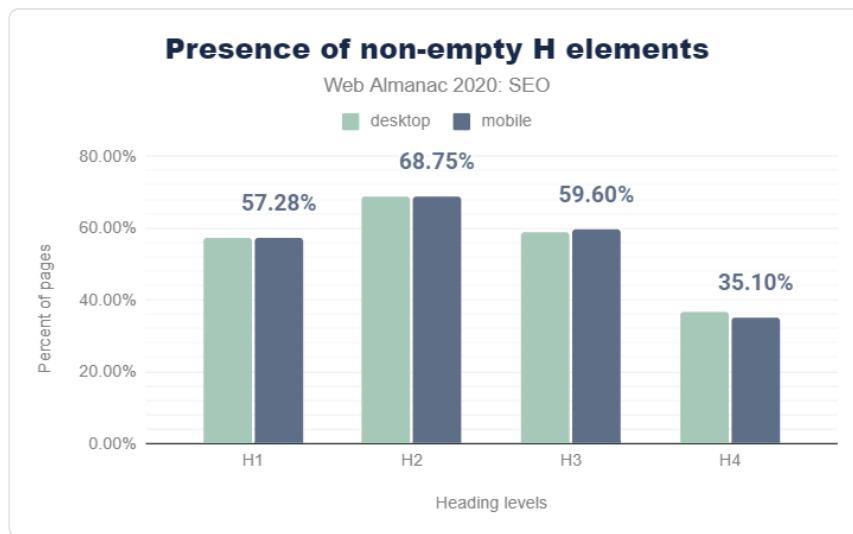


Figure 7.9. Usage of heading levels 1 through 4, excluding empty headings.

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

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

## Structured data

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

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

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

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

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

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

*Figure 7.10. Usage of each structured data format.*

We find that the disparity between mobile and desktop continues with this type of data. Microdata appeared on 19.55% of mobile pages and 17.94% of desktop pages. RDFa appeared on 1.42% of mobile pages and 1.63% of desktop pages.

### **Rendered versus non-rendered structured data**

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

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

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

### **Most prevalent structured data objects**

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

grow:

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

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

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

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

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

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

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

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

## Metadata

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

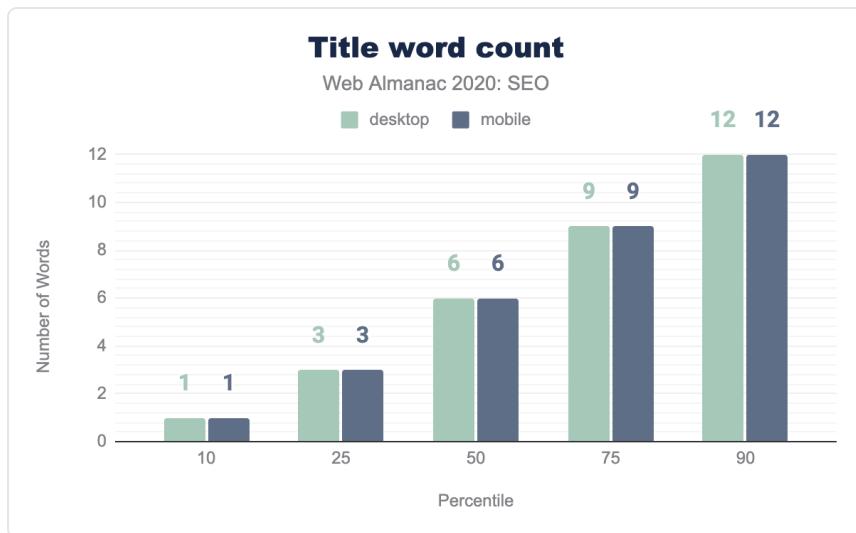
We examined these features to see how pages are quantitatively aligning with best practices to drive traffic from organic search.

## Titles

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

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

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



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

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

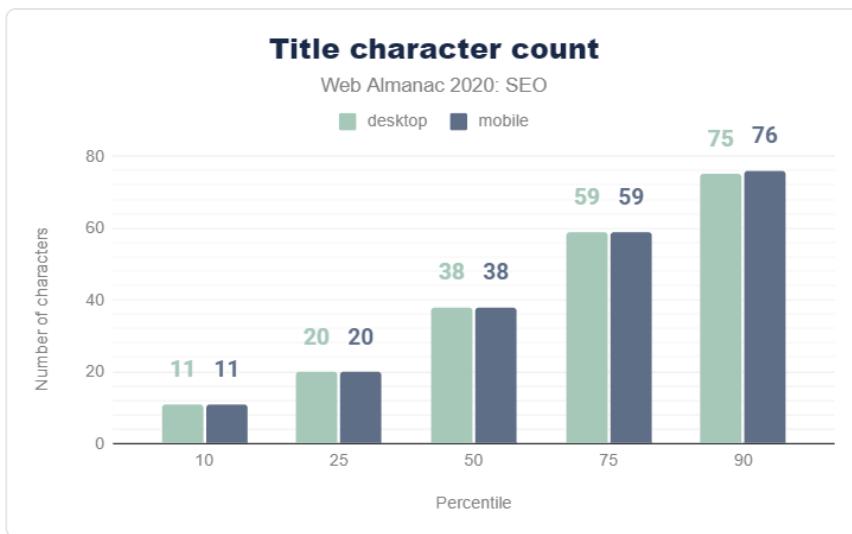


Figure 7.12. Distribution of the number of characters per page title.

## Meta descriptions

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

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

## Meta description word length

Web Almanac 2020: SEO

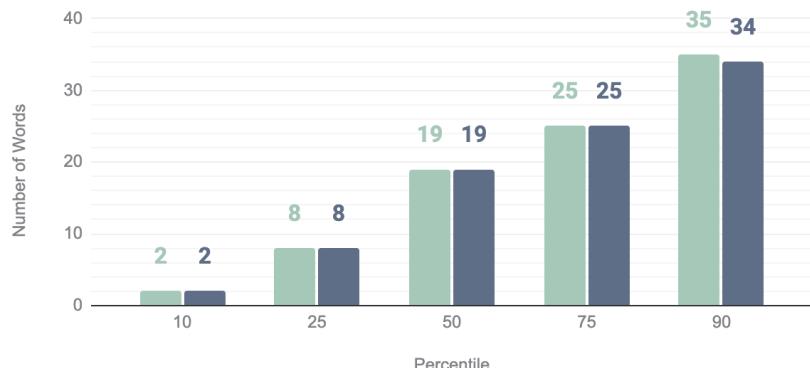
■ desktop    ■ mobile


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

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

## Meta description character length

Web Almanac 2020: SEO

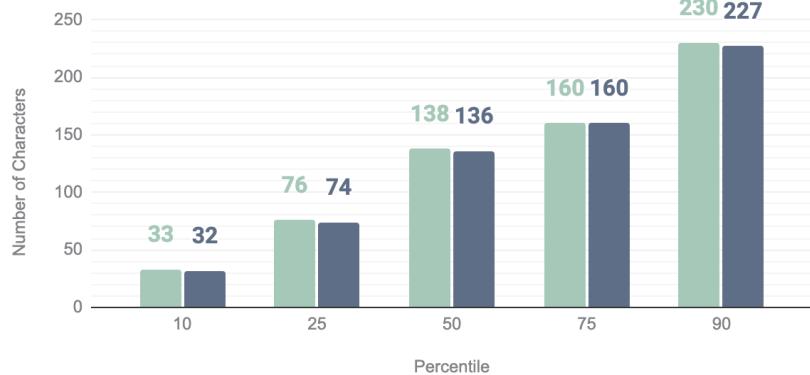
■ desktop    ■ mobile


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

The median character count for the meta description is 138 characters on desktop pages and 136 characters on mobile pages. Aside from the 75th percentile, there is a small disparity

between mobile and desktop meta description lengths distributed across the dataset. SEO best practices suggest limiting the specified meta description to up to 160 characters, but Google, inconsistently, may display upwards of 300 characters in its snippets.

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

## Images

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

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

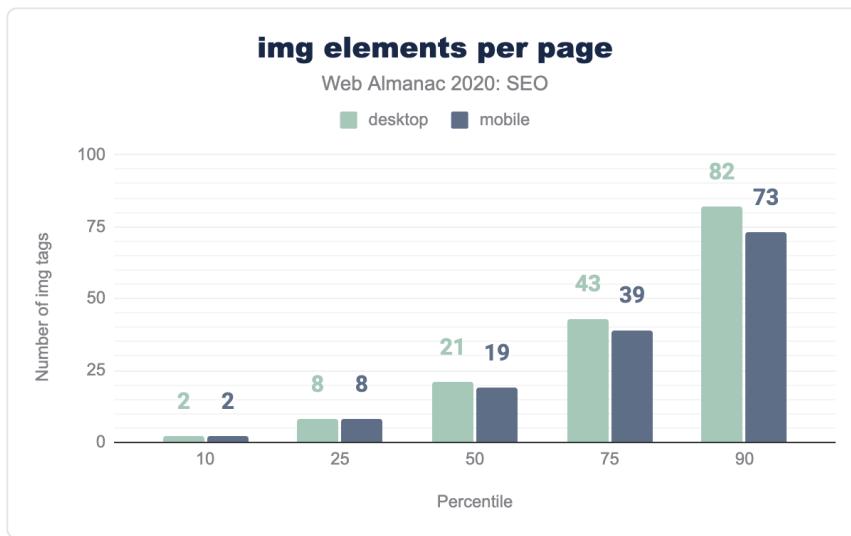


Figure 7.15. Distribution of the number of `<img>` elements per page.

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

Accessibility chapter.

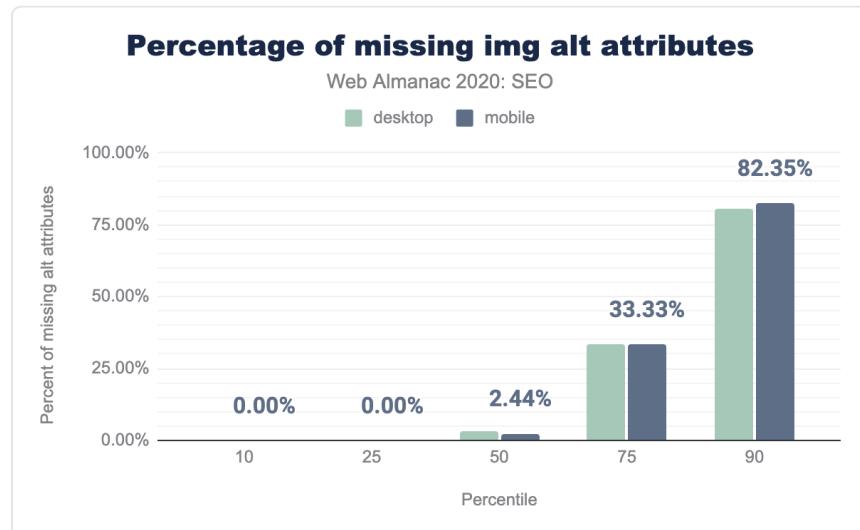


Figure 7.16. Distribution of the percent of `<img>` elements missing `image alt` attributes per page.

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

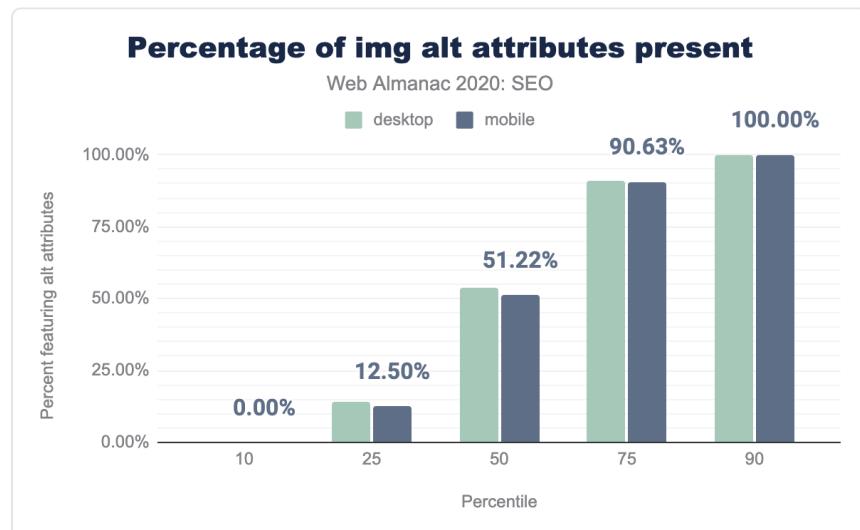


Figure 7.17. Distribution of the percent of images having `alt` attributes per page.

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

on mobile.

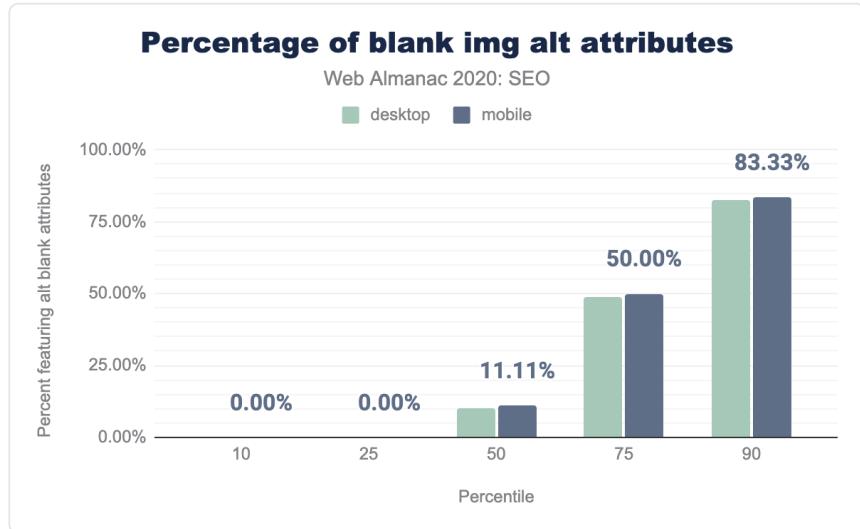


Figure 7.18. Distribution of the percent of images having blank `alt` attributes per page.

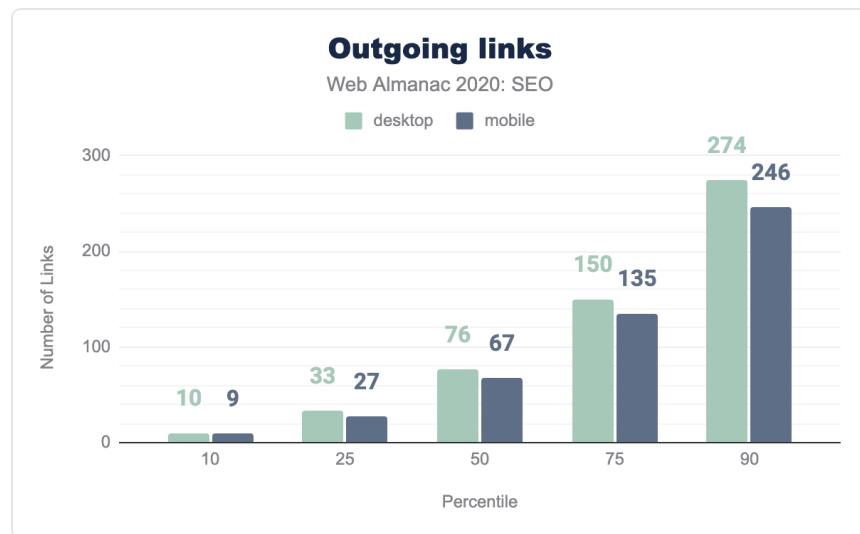
## Links

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

### Outgoing links

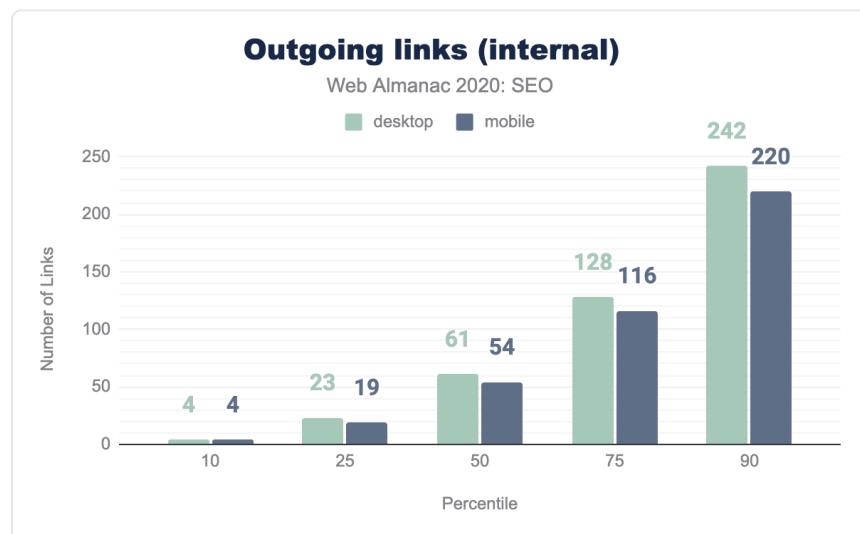
As part of this analysis we are able to assess the outgoing links from each page, whether to internal pages from the same domain, as well as external ones, however, have not analyzed incoming links.

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



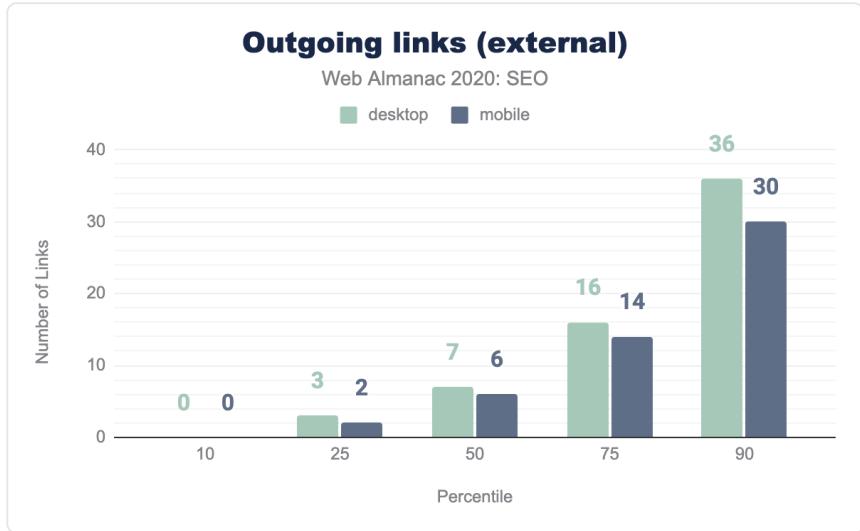
*Figure 7.19. Distribution of the number of links per page.*

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



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

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



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

## Text versus image links

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

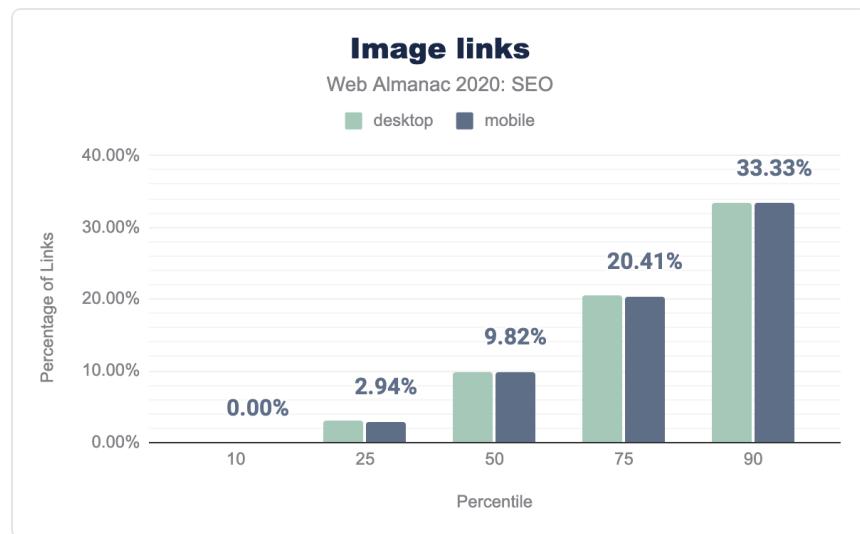


Figure 7.22. Distribution of the percent of links containing images per page.

### Mobile versus desktop links

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

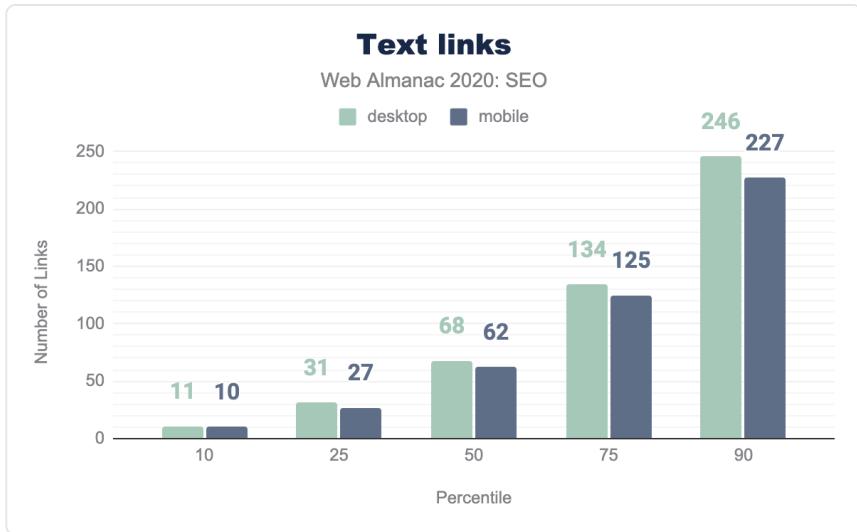


Figure 7.23. Distribution of the number of text links per page.

### **`rel=nofollow`, `ugc`, and `sponsored` attributes usage**

In September of 2019, Google introduced attributes that allow publishers to classify links as being sponsored or user generated content. These attributes are in addition to `rel=nofollow` which was previously introduced in 2005. The new attributes, `rel=ugc` and `rel=sponsored`, are meant to clarify or qualify the reason as to why these links are appearing on a given web page.

Our review of pages indicates that 28.58% of pages include `rel=nofollow` attributes on desktop and 30.74% on mobile. However, `rel=ugc` and `rel=sponsored` adoption is quite low with less than 0.3% of pages (about 20,000) having either. Since these attributes don't add any more value to a publisher than `rel=nofollow`, it is reasonable to expect that the rate of adoption will continue to be slow.

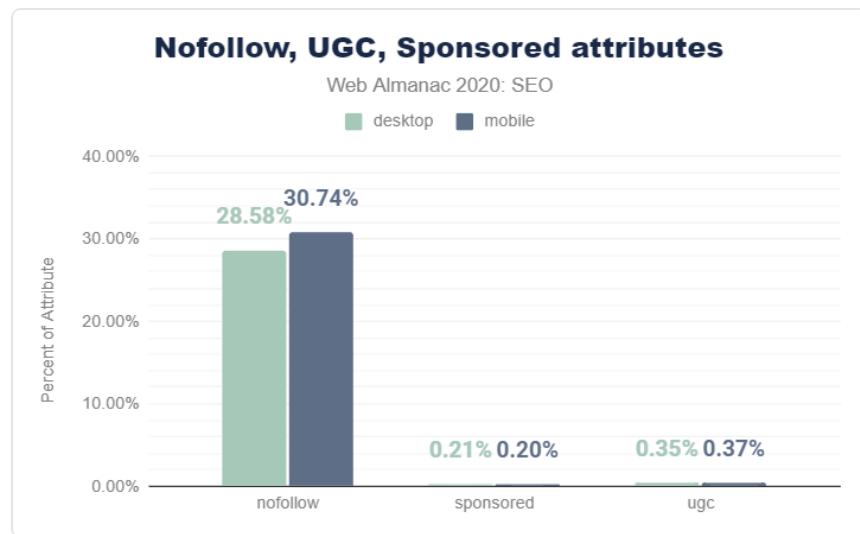


Figure 7.24. Percent of pages having `rel=nofollow`, `rel=ugc`, and `rel=sponsored` attributes.

## Advanced

This section explores the opportunities for optimization related to web configurations and elements that may not directly affect a site's crawlability or indexability, but have either been confirmed by search engines to be used as ranking signals or will affect the capacity of websites to leverage search features.

### Mobile friendliness

With the increasing popularity of mobile devices to browse and search across the web, search engines have been taking mobile friendliness into consideration as a ranking factor for several years.

Also, as mentioned before, since 2016 Google has been moving to a mobile-first index, meaning that the content that is crawled, indexed, and ranked is the one accessible to mobile users and the Smartphone Googlebot.

Additionally, since July 2019 Google is using the mobile-first index for all new websites and earlier in March, it announced that 70% of pages shown in their search results have already shifted over. It is now expected that Google fully switches to a mobile-first index in March 2021.

Mobile friendliness should be fundamental for any website looking to provide a good search experience—and as a consequence, grow in search results.

A mobile-friendly website can be implemented through different configurations: by using a responsive web design, with dynamic serving, or via a separate mobile web version. However, maintaining a separate mobile web version is not recommended anymore by Google, who endorse responsive web design instead.

## Viewport meta tag

The browser's viewport is the visible area of a page content, which changes depending on the used device. The `<meta name="viewport">` tag (or viewport meta tag) allows you to specify to browsers the width and scaling of the viewport, so that it is correctly sized across different devices. Responsive websites use the viewport meta tag as well as CSS media queries to deliver a mobile friendly experience.

42.98% of mobile pages and 43.2% desktop ones are have a viewport meta tag with the `content=initial-scale=1, width=device-width` attribute. However, 10.84% of mobile pages and 16.18% of desktop ones are not including the tag at all, suggesting that they are not yet mobile friendly.

Viewport	Mobile	Desktop
<code>initial-scale=1, width=device-width</code>	42.98%	43.20%
<code>not-set</code>	10.84%	16.18%
<code>initial-scale=1, maximum-scale=1, width=device-width</code>	5.88%	5.72%
<code>initial-scale=1, maximum-scale=1, user-scalable=no, width=device-width</code>	5.56%	4.81%
<code>initial-scale=1, maximum-scale=1, user-scalable=0, width=device-width</code>	3.93%	3.73%

Figure 7.25. Percent of pages having each viewport meta tag `content` attribute value.

## CSS media queries

Media queries are a CSS3 feature that play a fundamental role in responsive web design, as they allow you to specify conditions to apply styling only when the browser and device match certain rules. This allows you to create different layouts for the same HTML depending on the viewport size.

We found that 80.29% of desktop pages and 82.92% of the mobile ones are using either a `height`, `width`, or `aspect-ratio` CSS feature, meaning that a high percentage of pages have responsive features. The most popularly used features can be seen in the table below.

Feature	Mobile	Desktop
<code>max-width</code>	78.98%	78.33%
<code>min-width</code>	75.04%	73.75%
<code>-webkit-min-device-pixel-ratio</code>	44.63%	38.78%
<code>orientation</code>	33.48%	33.49%
<code>max-device-width</code>	26.23%	28.15%

Figure 7.26. Percent of pages that include each media query feature.

### Vary: User-Agent

When implementing a mobile friendly website with a dynamic serving configuration—one in which you show different HTMLs of the same page based on the used device—it is recommended to add a `Vary: User-Agent` HTTP header to help search engines discover the mobile content when crawling the website, as it informs that the response varies depending on the user agent.

Only 13.48% of the mobile pages and 12.6% of the desktop pages were found to specify a `Vary: User-Agent` header.

```
<link rel="alternate" media="only screen and (max-width: 640px)">
```

Desktop websites that have separate mobile versions are recommended to link to them using this tag in the `head` of their HTML. Only 0.64% of the analyzed desktop pages were found to be including the tag with the specified `media` attribute value.

### Web performance

Having a fast-loading website is fundamental to provide a great user search experience. Because of its importance, it has been taken into consideration as a ranking factor by search engines for years. Google initially announced using site speed as a ranking factor in 2010, and then in 2018 did the same for mobile searches.

As announced in November 2020, three performance metrics known as Core Web Vitals are on track to be a ranking factor as part of the "page experience" signals in May 2021. Core Web Vitals consist of:

### Largest Contentful Paint (LCP)

- Represents: user-perceived loading experience
- Measurement: the point in the page load timeline when the page's largest image or text block is visible within the viewport
- Goal: <2.5 seconds

### First Input Delay (FID)

- Represents: responsiveness to user input
- Measurement: the time from when a user first interacts with a page to the time when the browser is actually able to begin processing event handlers in response to that interaction
- Goal: <300 milliseconds

### Cumulative Layout Shift (CLS)

- Represents: visual stability
- Measurement: the sum of the number of *layout shift scores* approximating the percent of the viewport that shifted
- Goal: <0.10

## Core Web Vitals experiences per device

Desktop continues to be the more performant platform for users despite more users on mobile devices. 33.13% of websites scored *Good* Core Web Vitals for desktop while only 19.96% of their mobile counterparts passed the Core Web Vitals assessment.

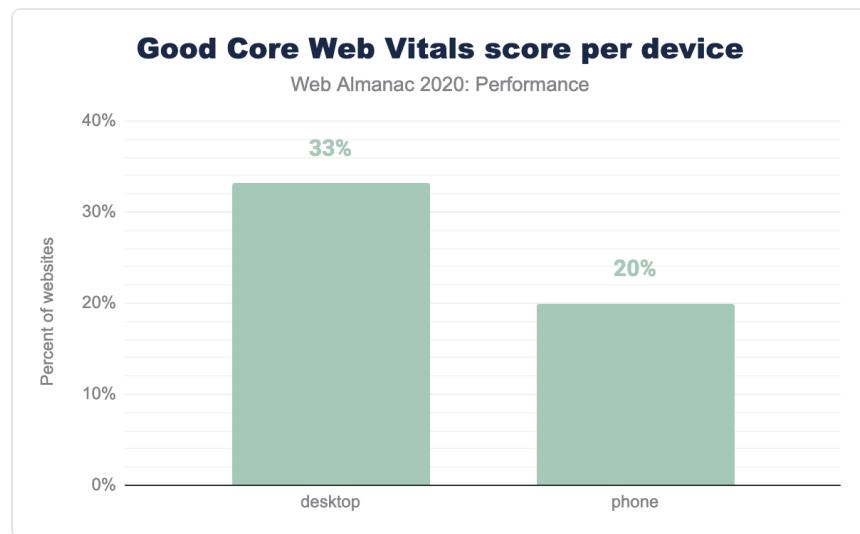


Figure 7.27. Percent of websites passing the Core Web Vitals assessment per device.

### Core Web Vitals experiences per country

A user's physical location impacts performance perception as their locally available telecom infrastructure, network bandwidth capacity, and the cost of data create unique loading conditions.

Users located in the United States recorded the largest absolute number of websites with Good Core Web Vitals experiences despite only 32% of sites earning the passing grade. Republic of Korea recorded the highest percentage of Good Core Web Vital experiences at 52%. The relative portion of total websites requested by each country is worth noting. Users in United States generated 8X the total origin requests as generated by Republic of Korea users.

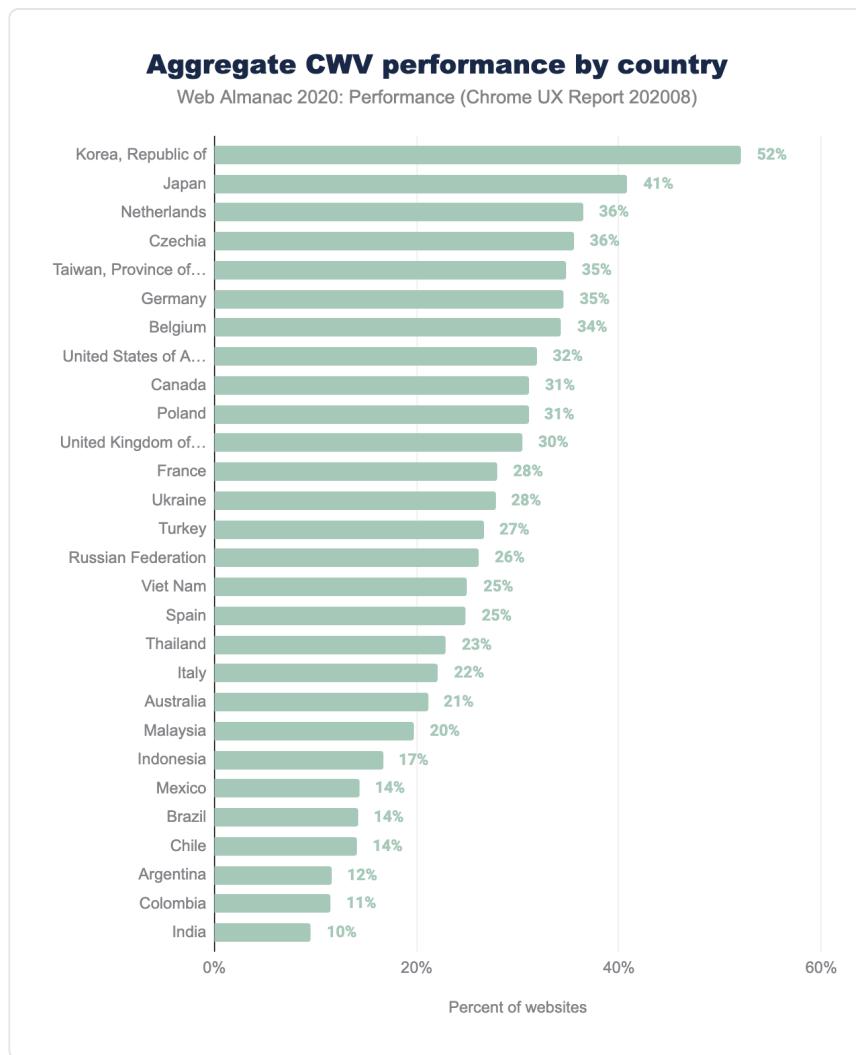


Figure 7.28. Percent of websites passing the Core Web Vitals assessment per country.

Additional analyses of Core Web Vitals performance by dimensions by effective connection type and specific metrics are available in the Performance chapter.

## Internationalization

Internationalization covers the configurations that multilingual or multi-country websites can use to inform search engines about their different language and/or country versions, specify

which are the relevant pages to show users in each case, and avoid targeting issues.

The two international configurations that we analyzed are the `content-language` meta tag and the `hreflang` attributes, that can be used to specify the language and the content of each page. Additionally, `hreflang` annotations allow you to specify the alternate language or country versions of each page.

Search engines like Google and Yandex use `hreflang` attributes as a signal to determine the page's language and country target, and although Google doesn't use the HTML lang or the `content-language` meta tag, the latter last tag is used by Bing.

### **hreflang**

8.1% of desktop pages and 7.48% of mobile pages use the `hreflang` attribute, which might seem low, but this is natural because these are only used by multilingual or multi-country websites.

We found that only 0.09% of the desktop pages and 0.07% of the mobile pages implement `hreflang` via their HTTP headers, and that most rely on the HTML `head` implementation.

We also identified that there are some pages that rely on JavaScript to render `hreflang` annotations. 0.12% of desktop and mobile pages are showing `hreflang` in the rendered but not in the raw HTML.

From a language and country value perspective, when analyzing the implementation via the HTML head, we found that English (`en`) is the most popular used value, with 4.11% of the mobile and 4.64% of the desktop pages including it. After English, the second most popular value is `x-default` (used when defining a `default` or `fallback` version for users of non-targeted languages or countries), with 2.07% of mobile and 2.2% of the desktop pages including it.

The third, fourth and fifth most popular are German (`de`), French (`fr`) and Spanish (`es`), followed by Italian (`it`) and English for the US (`en-us`), as can be seen in the table below with the rest of the values implemented via the HTML head.

<b>Value</b>	<b>Mobile</b>	<b>Desktop</b>
<code>en</code>	4.11%	4.64%
<code>x-default</code>	2.07%	2.20%
<code>de</code>	1.76%	1.88%
<code>fr</code>	1.74%	1.87%
<code>es</code>	1.74%	1.84%
<code>it</code>	1.27%	1.33%
<code>en-us</code>	1.15%	1.31%
<code>ru</code>	1.12%	1.13%
<code>en-gb</code>	0.87%	0.98%
<code>pt</code>	0.87%	0.87%
<code>nl</code>	0.83%	0.94%
<code>ja</code>	0.73%	0.81%
<code>pl</code>	0.72%	0.75%
<code>de-de</code>	0.69%	0.78%
<code>tr</code>	0.69%	0.66%

Figure 7.29. Percent of pages that include the top `hreflang` values in the HTML head.

Something slightly different was found in top `hreflang` language and country values implemented via the HTTP headers, with English (`en`) being again the most popular one, although in this case followed by French (`fr`), German (`de`), Spanish (`es`) and Dutch (`nl`) as the top values.

<b>Values</b>	<b>Mobile</b>	<b>Desktop</b>
en	0.05%	0.06%
fr	0.02%	0.02%
de	0.01%	0.02%
es	0.01%	0.01%
nl	0.01%	0.01%

Figure 7.30. Percent of pages that include the top `hreflang` values in HTTP headers.

## Content-Language

When analyzing the `content-language` usage and values, whether by implementing it as a meta tag in the `HTML head` or in the HTTP headers, we found that only 8.5% of mobile pages and 9.05% of desktop pages were specifying it in the HTTP headers. Even fewer websites were specifying their language or country with the `content-language` tag in the `HTML head`, with only 3.63% of mobile pages and 3.59% of desktop pages featuring the meta tag.

From a language and country value perspective, we found that the most popular values specified in the `content-language` meta-tag and HTTP headers are English (`en`) and English for the US (`en-us`).

In the case of English (`en`) we identified that 4.34% of desktop and 3.69% of mobile pages specified it in the HTTP headers and 0.55% of the desktop and 0.48% of the mobile pages were doing so via the `content-language` meta tag in the `HTML head`.

For English for the US (`en-us`), the second most popular value, it was found that only 1.77% of mobile pages and 1.7% of desktop ones were specifying it in the HTTP headers, and 0.3% of the mobile pages and 0.36% desktop ones were doing it so in the HTML.

The rest of the most popular language and country values can be seen in the tables below.

<b>Value</b>	<b>Mobile</b>	<b>Desktop</b>
<i>en</i>	3.69%	4.34%
<i>en-us</i>	1.77%	1.70%
<i>de</i>	0.50%	0.44%
<i>es</i>	0.34%	0.33%
<i>fr</i>	0.31%	0.34%
<i>ru</i>	0.18%	0.16%
<i>pt-br</i>	0.15%	0.16%
<i>nl</i>	0.13%	0.15%
<i>it</i>	0.13%	0.13%
<i>ja</i>	0.08%	0.10%

Figure 7.31. Percent of pages using the top `content-language` values in HTTP headers.

<b>Value</b>	<b>Mobile</b>	<b>Desktop</b>
<i>en</i>	0.48%	0.55%
<i>en-us</i>	0.30%	0.36%
<i>pt-br</i>	0.24%	0.24%
<i>ja</i>	0.19%	0.26%
<i>fr</i>	0.18%	0.19%
<i>tr</i>	0.17%	0.13%
<i>es</i>	0.16%	0.15%
<i>de</i>	0.15%	0.11%
<i>cs</i>	0.12%	0.12%
<i>pl</i>	0.11%	0.09%

Figure 7.32. Percent of pages using the top `content-language` values in HTML meta tags.

## Security

Google places a specific focus on security in all respects. The search engine maintains lists of sites that have shown suspicious activity or have been hacked. Search Console surfaces these issues and Chrome users are presented with warnings before visiting sites with these problems. Additionally, Google provides an algorithmic boost to pages that are served over HTTPS (Hypertext Transfer Protocol Secure). For a more in-depth analysis on this topic, see the Security chapter.

### HTTPS usage

We found that 77.44% of desktop pages and 73.22% of mobile pages have adopted HTTPS. This is up 10.38% from last year. It is important to note that browsers have become more aggressive in pushing HTTPS by signaling that pages are insecure when you visit them without HTTPS. Also, HTTPS is currently a requirement to capitalize on higher performing protocols such as HTTP/2 and HTTP/3 (also known as HTTP over QUIC). You can learn more about the state of these protocols in the HTTP/2 chapter.

All of these things have likely contributed to the higher adoption rate year over year.

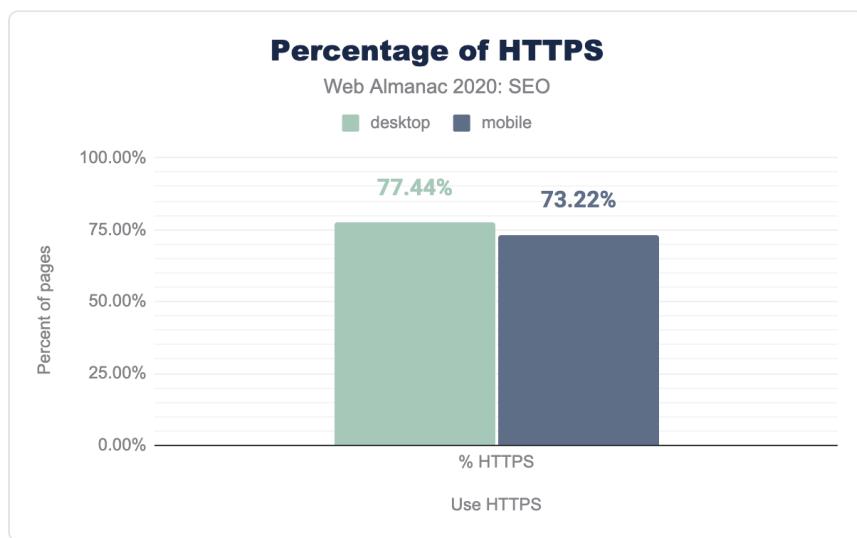


Figure 7.33. Percent of pages served with HTTPS.

## AMP

AMP (previously called Accelerated Mobile Pages) is an open source HTML framework that

was launched by Google in 2015 as a way to help pages load more quickly, especially on mobile devices. AMP can be implemented as an alternate version of existing web pages or developed from scratch using the AMP framework.

When there's an AMP version available for a page, it will be shown by Google in mobile search results, along with the AMP logo.

It is also important to note that while AMP usage is not a ranking factor for Google (or any other search engine), web speed is a ranking factor.

Additionally, as of this writing, AMP is a requirement to be featured in Google's Top Stories carousel in mobile search results, which is an important feature for news-related publications. However, this will change in May 2021, when non-AMP content will become eligible as long as it meets the Google News content policies and provides a great page experience as announced by Google in November this year.

When checking the usage of AMP as an alternate version of a non-AMP based page, we found that 0.69% of mobile web pages and 0.81% of desktop ones were including an `<amphtml>` tag pointing to an AMP version. Although the adoption is still very low, this is a slight improvement from last year's findings, in which only 0.62% of mobile pages contained a link to an AMP version.

On the other hand, when assessing the usage of AMP as a framework to develop websites, we found that only 0.18% of mobile pages and 0.07% of desktop ones were specifying the `<html amp>` or `<html⚡>` emoji attribute, which are used to indicate AMP-based pages.

## Single-page applications

Single-page applications (SPAs) enable browsers to retain and update a single page load even as the on-page content updates to match a user request. Multiple technologies such as JavaScript frameworks, AJAX, and Websockets are used to accomplish lightweight subsequent page loads.

These frameworks required special SEO considerations, although Google has worked to mitigate the issues caused by client-side rendering with aggressive caching strategies. In a video from Google Webmaster's 2019 conference, Software Engineer Erik Hendriks shared that Google no longer relies on `Cache-Control` headers and instead looks for `ETag` or `Last-Modified` headers to see if the content of the file has changed.

SPAs should utilize the Fetch API for granular control of caching. The API allows for the passing of `Request` objects with specific cache overrides set and can be used to set the necessary `If-Modified-Since` and `ETag` headers.

Undiscoverable resources are still the primary concern of search engines and their web crawlers. Search crawlers look for the `href` attributes in `<a>` tags to find linked pages. Without

these, the page is seen as isolated without internal linking. 5.59% of desktop pages studied contained no internal links as well as 6.04% of mobile-rendered pages. This is a marker that the page is part of a JavaScript framework SPA and missing the necessary `<a>` tag with valid `href` attributes required for their internal linking to be discovered.

The discoverability of links in popular JavaScript frameworks used for SPAs increased dramatically in 2020 over the previous year. In 2019, 13.08% of mobile navigation links on React sites used deprecated hash URLs. For 2020, only 6.12% of the tested React links were hashed.

Similarly, Vue.js saw a drop to 3.45% from the previous year's 8.15%. Angular was the least likely to use uncrawlable hashed mobile navigation links in 2019 with only 2.37% of mobile pages using them. For 2020, that number plummeted to 0.21%.

## Conclusion

Consistent with what was found and concluded last year, most sites have crawlable and indexable desktop and mobile pages, and are making use of the fundamental SEO-related configurations.

It is important to highlight how the link discoverability for major JavaScript frameworks used for SPAs increased dramatically compared to 2019. By testing mobile navigation links for hashed URLs, we saw -53% instances of uncrawlable links from sites using React, -58% fewer from Vue.js powered sites, and a -91% reduction from Angular SPAs.

Additionally, we also identified that there has been a slight improvement from last year's findings across many of the analyzed areas:

- **`robots.txt`:** Last year 72.16% of mobile sites had a valid `robots.txt` versus 74.91% this year.
- **`canonical` tag:** Last year we identified that 48.34% of mobile pages were using a canonical tag versus 53.61% this year.
- **`title` tag:** This year we found that 98.75% of the desktop pages are featuring one, while 98.7% of mobile pages are also including it. Last year's chapter found that 97.1% of mobile pages were featuring a `title` tag.
- **`meta description`:** This year, we found 68.62% of desktop pages and 68.22% of mobile ones had a `meta description`, an improvement from last year when 64.02% of mobile pages had one.
- **structured data:** Despite the fact that reviews are not supposed to be associated with home pages, the data indicates that `AggregateRating` is up 23.9% on mobile and 23.7% on desktop.
- **HTTPS usage:** 77.44% of desktop pages and 73.22% of mobile pages have adopted HTTPS. This is up 10.38% from last year.

However, not everything has improved over the last year. The median desktop page includes 61 internal links while the median mobile page has 54. This is down 12.8% and 10% respectively from last year, suggesting that sites are not maximizing the ability to improve the crawlability and link equity flow through their pages.

It is also important to note how there's still an important opportunity for improvement across many critical SEO related areas and configurations. Despite the growing use of mobile devices and Google's move to a mobile-first index:

- 10.84% of mobile pages and 16.18% of desktop ones are not including the `viewport` tag at all, suggesting that they are not yet mobile friendly.
- Non-trivial disparities were found across mobile and desktop pages, like the one between mobile and desktop links, illustrated in the 62 links on mobile versus the 68 links on desktop for the median web page.
- 33.13% of websites scored *Good* Core Web Vitals for desktop while only 19.96% of their mobile counterparts passed the Core Web Vitals assessment, suggesting that desktop continues to be the more performant platform for users.

These findings could negatively impact sites as Google completes its migration to a mobile-first index in March 2021.

Disparities were also found across rendered and non-rendered HTML. For example, the median mobile page displays 11.5% more words when rendered than its raw HTML, indicating a reliance on client-side JavaScript to show content.

Search crawlers look for the `<a href>` tags to find linked pages. Without these, the page is seen as isolated without internal linking. 5.59% of desktop pages contained no internal links as well as 6.04% of mobile-rendered pages.

These findings suggest that search engines are continually evolving in their capacity to effectively crawl, index, and rank websites, and some of the most important SEO configurations are now also better taken into consideration.

However, many sites across the web are still missing out on important search visibility and growth opportunities, which also shows the persisting need of SEO evangelization and best practices adoption across organizations.

## Authors

---



### Aleyda Solis

Twitter: @aleyda | GitHub: aleyda | Website: <http://www.aleydasolis.com/en/>

SEO consultant, author, speaker and entrepreneur. Founder of Orainti<sup>13</sup> (a boutique SEO consultancy working with some of the top Web properties and brands, from SaaS to marketplaces) and co-founder of Remoters.net<sup>14</sup> (a free remote work hub, featuring remote jobs, tools, events, guides, and more to facilitate remote work). Aleyda enjoys sharing about SEO through her blog<sup>15</sup>, the #SEOFOMO newsletter<sup>16</sup>, the Crawling Mondays<sup>17</sup> video and podcast series and over Twitter<sup>18</sup>.



### Michael King

Twitter: @IPullRank | GitHub: ipullrank

An artist and a technologist, all rolled into one, Michael King is the founder of enterprise digital marketing agency, iPullRank<sup>19</sup> and founder of Natural Language Generation app CopyScience<sup>20</sup>. Effortlessly leaning on his background as an independent hip-hop musician, Mr. King is a compelling content creator and an award-winning dynamic speaker who is called upon to contribute to technology and marketing conferences and blogs all over the world. You can find Mike talking trash on Twitter<sup>21</sup>.



### Jamie Indigo

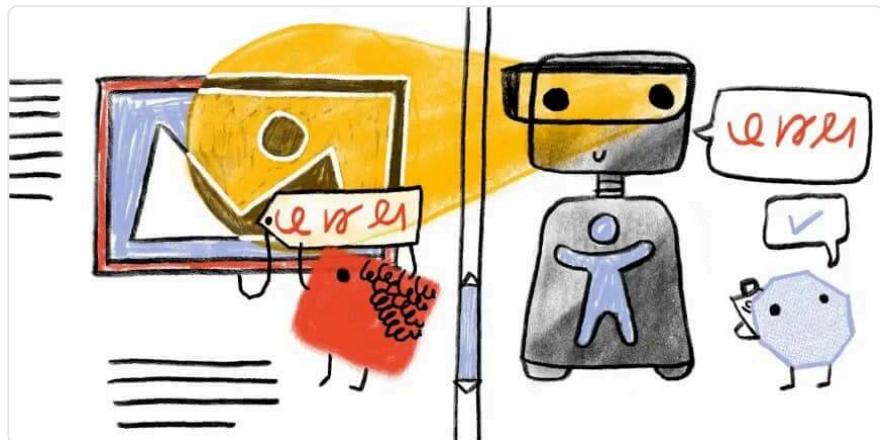
Twitter: @Jammer\_Volts | GitHub: fellowhuman1101 | Website: <https://not-a-robot.com/>

100% human & totally not a robot, Jamie Indigo untangles technologies to help humans access useful information and businesses provide valuable digital experiences. She founded Not a Robot<sup>22</sup> to consult with a focus on the human aspects of technical SEO including ethics & inclusion in technology and the search industry. She can found be learning in public on Twitter<sup>23</sup>.

- 
- 13. <https://www.orainti.com/>
  - 14. <https://remoters.net/>
  - 15. <https://www.aleydasolis.com/en/blog/>
  - 16. <https://www.aleydasolis.com/en/seo-tips/>
  - 17. <https://www.aleydasolis.com/en/crawling-mondays-videos/>
  - 18. <https://twitter.com/aleyda>
  - 19. <https://ipullrank.com>
  - 20. <https://www.copyscience.io>
  - 21. <https://twitter.com/ipullrank>
  - 22. <https://not-a-robot.com>
  - 23. [https://twitter.com/Jammer\\_Volts](https://twitter.com/Jammer_Volts)

# Part II Chapter 8

# Accessibility [UNEDITED]



Written by Olu Niyi-Awosusi and Alex Tait

Reviewed by Adrian Roselli, Eric Bailey, and David Fox

Analyzed by David Fox

## Authors



Olu Niyi-Awosusi

👤 oluoluoxenfree 🌐 <http://www.opentagclosetag.com/>



Alex Tait

🐦 @at\_fresh\_dev 🏙 alextait1 🌐 <https://togethertech.ca/>



# Part II Chapter 9

# Performance



*Written by Karolina Szczur*

*Reviewed by Boris Schapira, Rick Viscomi, David Fox, Noam Rosenthal, and Leonardo Zizzamia*

*Analyzed by Max Ostapenko and Pokidov N. Dmitry*

## Introduction

There is an unquestionable, detrimental effect that slow speed has on overall user experience, and consequently, conversions. But poor performance doesn't just cause frustration or negatively affects business goals—it creates real-life barriers to entry. This year's global pandemic made those existing barriers even more apparent. With the shift to remote learning, work and socializing, suddenly our entire lives were moved online. Poor connectivity and lack of access to capable devices made this change painful at best, if not impossible, to many. It has been a real test, highlighting connectivity, device and speed inequalities worldwide.

Performance tooling continues to evolve to portray those diverse aspects of user experience and make it easier to find underlying issues. Since last year's Performance chapter, there have been numerous significant developments in the space that have already transformed how we approach speed monitoring.

With a significant release of the popular quality auditing tool, Lighthouse 6, the algorithm behind the famous Performance Score has changed significantly, and so did the scores. Core

Web Vitals, a set of new metrics portraying different aspects of user experience, has been released. It will be one of the factors for search ranking in the future, shifting the eyes of the development community onto the new speed signals.

In this chapter, we will be looking at real-world performance data provided by the Chrome User Experience Report (CrUX) through the lens of those new developments as well as analyzing a handful of other relevant metrics. It is important to note that due to the limitations of iOS, CrUX mobile results don't include devices running Apple's mobile operating system. This fact will undeniably affect our analysis, especially when examining metric performance on a per-country basis.

Let's dive in.

## Lighthouse Performance Score

In May 2020, Lighthouse 6 was released. The new major version of the popular performance auditing suite introduced notable changes to its Performance Score algorithm. The Performance Score is a high-level portrayal of site speed. In Lighthouse 6, the score is measured with six—not five—metrics: First Meaningful Paint and First CPU Idle were removed and replaced with Largest Contentful Paint (LCP), Total Blocking Time (TBT, the lab equivalent of First Input Delay) and Cumulative Layout Shift (CLS).

The new scoring algorithm is prioritizing the new generation of performance metrics: Core Web Vitals and deprioritizing First Contentful Paint (FCP), Time to Interactive (TTI) and Speed Index, as their score weight decreases. The algorithm also now distinctly emphasizes three aspects of user experience: *interactivity* (Total Blocking Time and Time to Interactive), *visual stability* (Cumulative Layout Shift) and *perceived loading* (First Contentful Paint, Speed Index, Largest Contentful Paint).

Additionally, the score is now calculated using different reference points for desktop and mobile. What this means in practice is that it will be less forgiving on desktop (expecting fast websites) and more relaxed on mobile (since benchmark performance on mobile is less quick than desktop). You can compare your Lighthouse 5 and 6 score difference in the Lighthouse Score calculator. So, how did the scores really change?

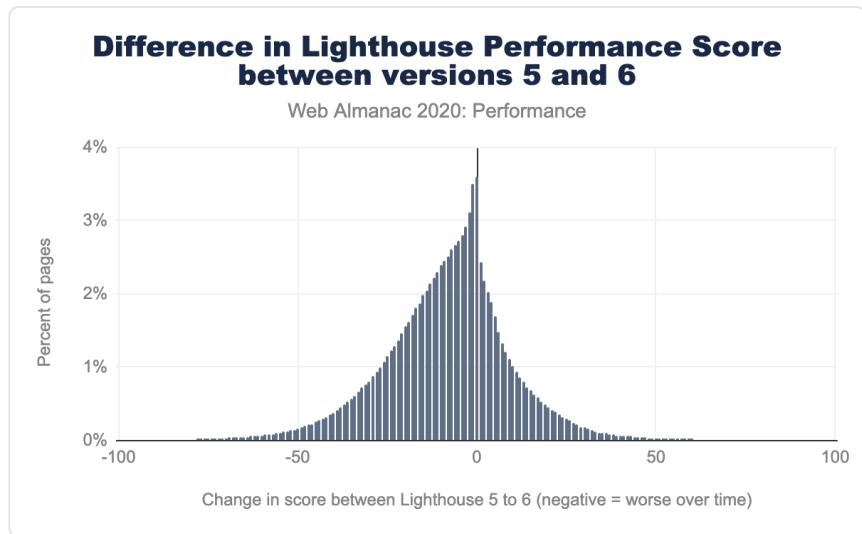


Figure 9.1. Difference in Lighthouse Performance Score between versions 5 and 6.

Above, we observe that 4% of websites recorded no Performance Score change, but the number of sites with negative changes outweighs the ones with score improvements. The Performance Score grades have gotten worse (with the most prominent decrease curve in the 10-25 points area), which is portrayed even more directly below:



Figure 9.2. Lighthouse Performance Score distribution for Lighthouse 5 and 6.

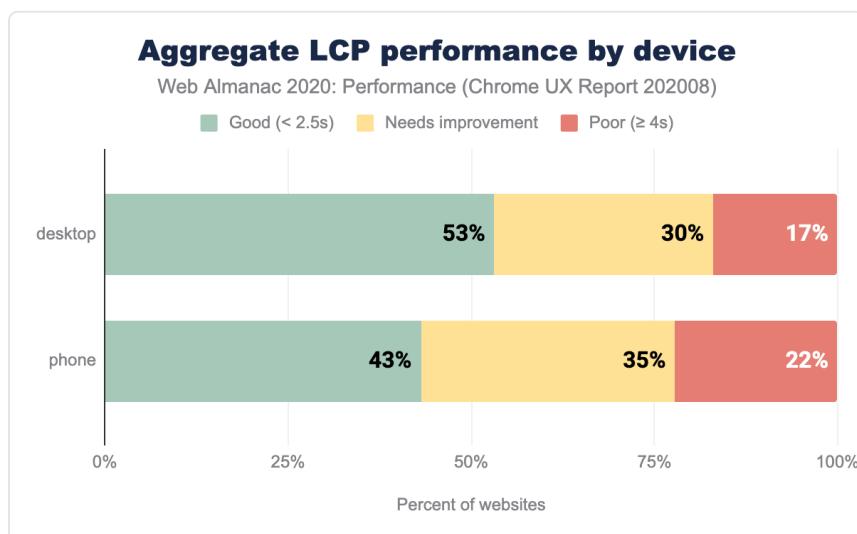
In the comparison of Lighthouse 5 and Lighthouse 6, we can directly observe how the distribution of score has changed. With the Lighthouse 6 algorithm, we observe a rise in the percentage of pages receiving scores between 0 to 25 and a decline between 50 and a 100. While in Lighthouse 5, we saw 3% of pages receiving 100 scores, on Lighthouse 6, that number fell to only 1%.

These overall changes are not unexpected considering a multitude of amendments to the algorithm itself.

## Core Web Vitals: Largest Contentful Paint

Largest Contentful Paint (LCP) is a landmark timing-based metric that reports the time at which the largest above-the-fold element was rendered.

### LCP by device



*Figure 9.3. Aggregate LCP performance split by device type.*

In the chart above, we can observe that between 43% and 53% of websites have good LCP performance (below 2.5s): the majority of websites manage to prioritize and load their critical, above-the-fold media fast. For a relatively new metric, this is an optimistic signal. The slight variance between desktop and mobile is likely to be caused by varying network speeds, device capabilities and image sizing (large, desktop-specific images will take longer to be downloaded and rendered).

## LCP by geographic location

### Aggregate LCP performance by country

Web Almanac 2020: Performance (Chrome UX Report 202008)

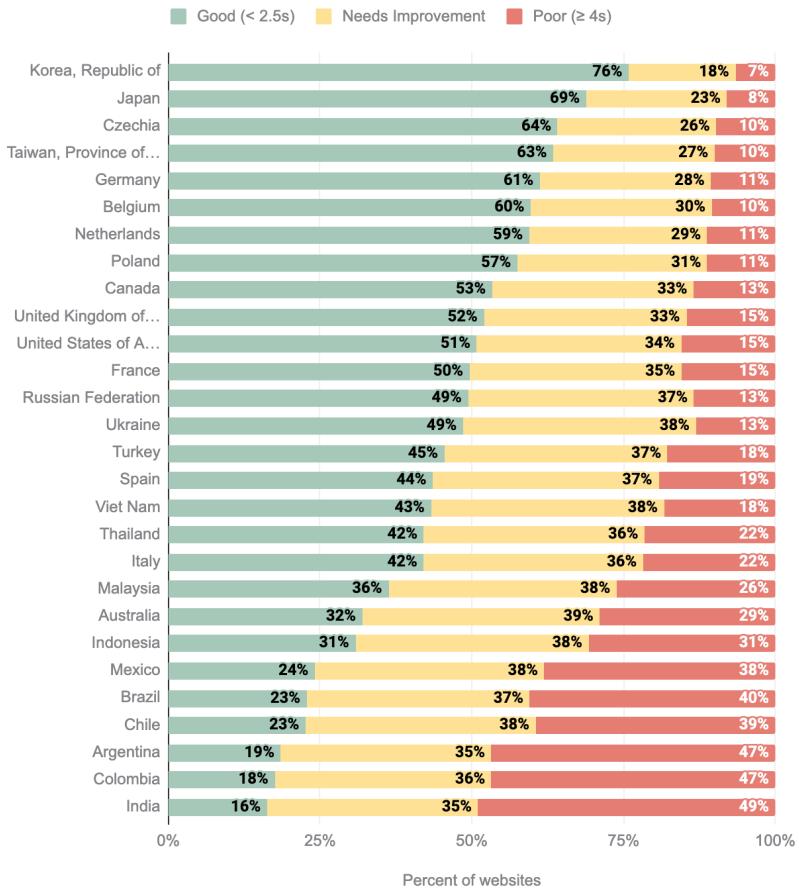


Figure 9.4. Aggregate LCP performance split by country.

The highest percentage of good LCP readings is mostly distributed amongst European and Asian countries with the Republic of Korea (South Korea) leading at 76% of good metric readings. South Korea is a consistent leader in mobile speeds, with an impressive 145 Mbps download reported by Speedtest Global Index for October. Japan, Czechia, Taiwan, Germany and Belgium are also a handful of countries with reliably high mobile speeds. Australia, while

leading in mobile network speeds, is let down by slow desktop connections and latency which places it at the bottom section of the ranking above.

India remains the last one in our set of data, with only 16% of good experiences. While the population of people connecting to the internet for the first time is continually growing, the mobile and desktop network speeds are still an issue, with average downloads of 10Mbps for 4G, 3Mbps for 3G and below 50Mbps for desktop.

## LCP by connection type

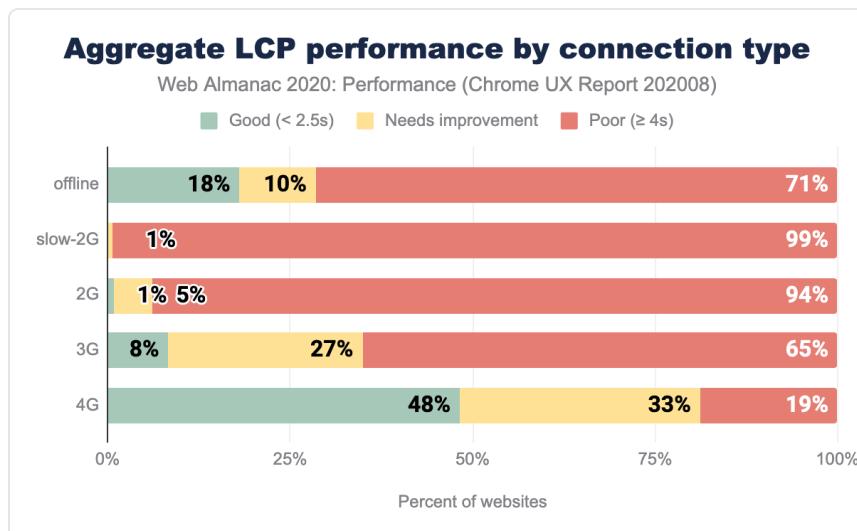


Figure 9.5. Aggregate LCP performance split by connection type.

Since LCP is a timing showcasing when the largest above-the-fold element has been rendered (including imagery, videos or block-level elements containing text), it is not surprising that the slower the network, the more significant portion of measurements are poor.

There's a clear correlation of network speed and better LCP performance, but even on 4G, only 48% of results are categorized as good, leaving half of the readings in need of an improvement. Automating media optimization, serving the right dimensions and formats, as well as optimizing for Low Data Mode, could help move the needle. Learn more about optimizing LCP in this guide.

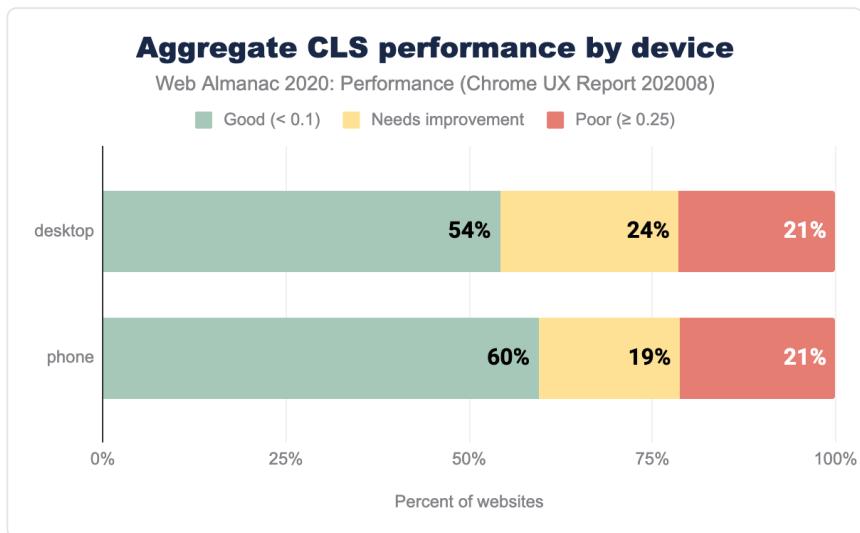
## Core Web Vitals: Cumulative Layout Shift

Cumulative Layout Shift (CLS) quantifies how much elements within the viewport move around

during the page visit. It helps pinpoint the degree to which unexpected movement occurs on your websites to grade the user experience, rather than attempting to measure a specific part of interaction with the help of a unit like seconds or milliseconds.

In that way, CLS is a different, new type of a UX holistic metric in comparison to others mentioned in this chapter.

## CLS by device



*Figure 9.6. Aggregate CLS performance split by device type.*

According to CrUX data, both in cases of desktop and mobile devices, more than half of the websites have a good CLS score. There's a slight difference (6 percentage points) between the number of good-rated websites between desktop and mobile, favoring the latter. We could speculate that phones lead in good CLS ratings due to mobile-optimized experiences that tend to be less feature and visuals-rich.

## CLS by geographic location

### Aggregate CLS performance by country

Web Almanac 2020: Performance (Chrome UX Report 202008)

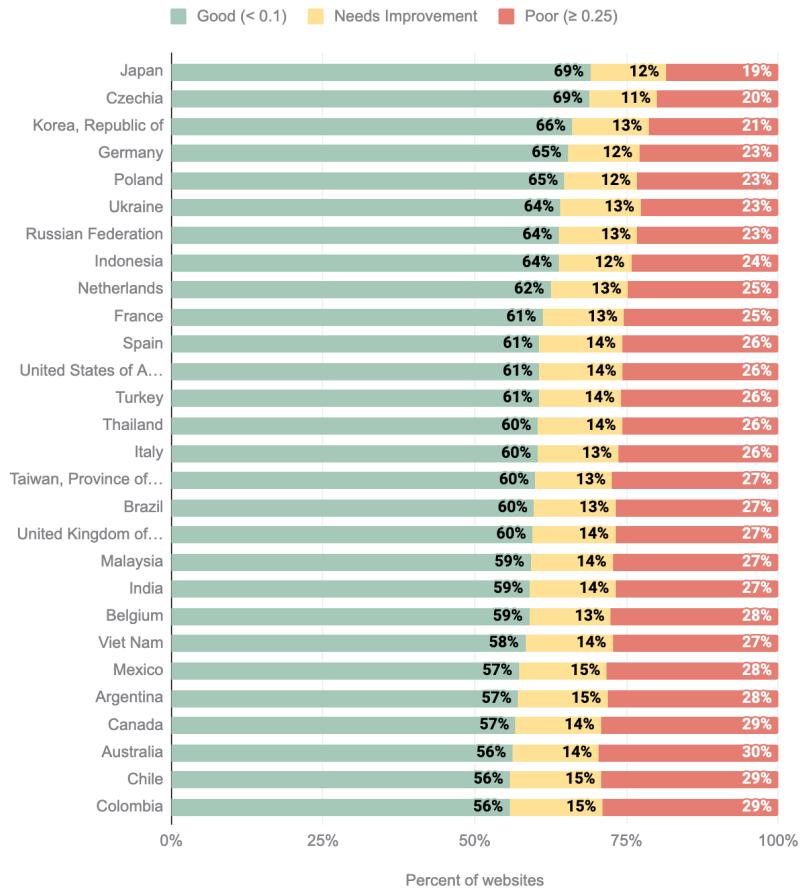


Figure 9.7. Aggregate CLS performance split by country.

The CLS performance in different geographical regions is primarily good, with at least 56% of sites with a good rating. This is excellent news for perceived visual stability. We can observe similar countries leading as we've seen in the LCP geo-distribution—Republic of Korea, Japan, Czechia, Germany, Poland.

Visual stability is less affected by geography and latency to other metrics, like LCP. The

difference in the percentage of good scores between the top and the bottom country is 61% for LCP and only 13% for CLS. The percentage of moderate-rating websites is relatively low across the board, giving way to 19%-29% of poor experiences across the board. There are numerous factors that can contribute to poor CLS—learn how to address them in the Optimize Cumulative Layout Shift guide.

## CLS by connection type

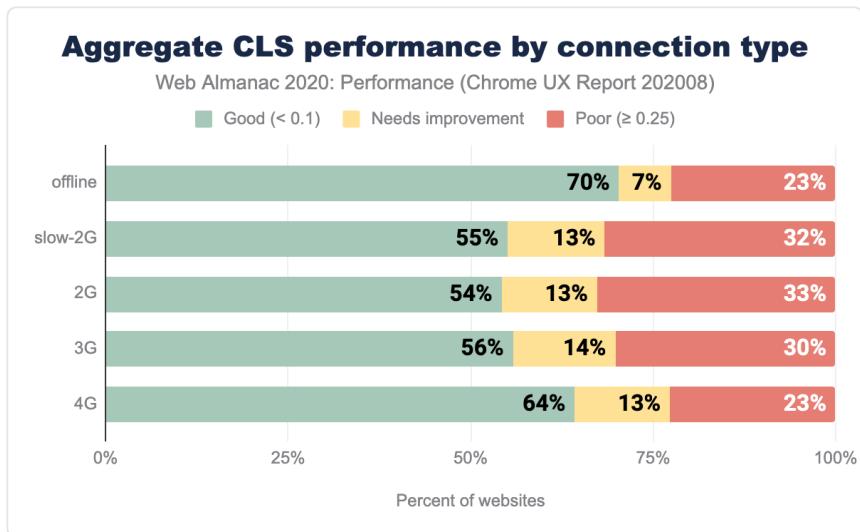


Figure 9.8. Aggregate CLS performance split by connection type.

There's a reasonably even distribution of CLS scoring across most connection types except for offline and 4G. In the offline scenario, we can speculate that Service Workers serve websites. Consequently, there's no delay in download caused by connection type, resulting in the most significant portion of good grades.

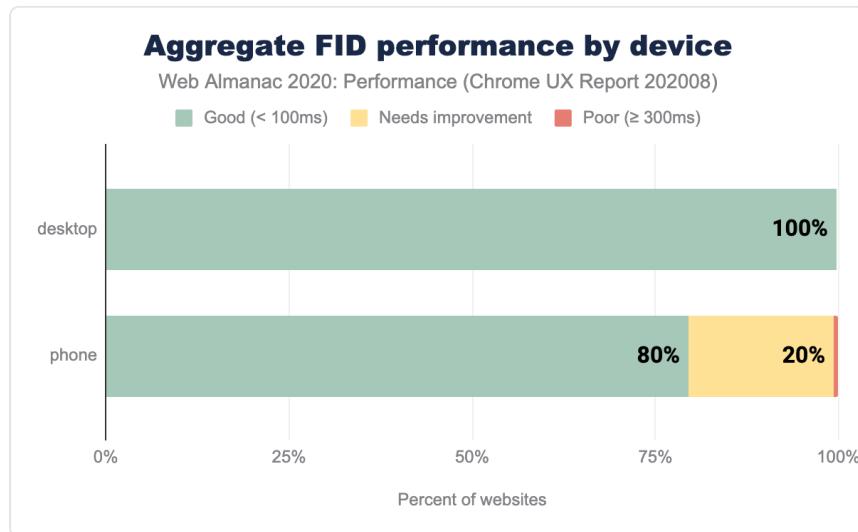
It is challenging to draw definite conclusions about 4G, but we can speculate that perhaps 4G+ connections are used as a method of internet access on desktop devices. If that assumption was valid, web fonts and imagery could be heavily cached, which positively affects CLS measurements.

## Core Web Vitals: First Input Delay

First Input Delay (FID) measures the time between first user interaction and when the browser is able to respond to that interaction. FID is a good indicator of how interactive your websites

are.

## FID by device



*Figure 9.9. Aggregate FID performance split by device type.*

It is relatively uncommon to see good scores distributed across such a high percentage of websites. On desktop, based on the 75th percentile of sites' distributions, 100% of sites report fast timings for FID, meaning the number of people experiencing interaction delays is very low.

On mobile, 80% of sites are graded as good. A likely explanation is the reduced CPU capabilities in comparison to desktop, network latency on mobile (causing a delay in script download and execution) as well as battery efficiency and temperature limitations, capping the CPU potential of mobile devices. All of which directly affect interactivity metrics like FID.

## FID by geographic location

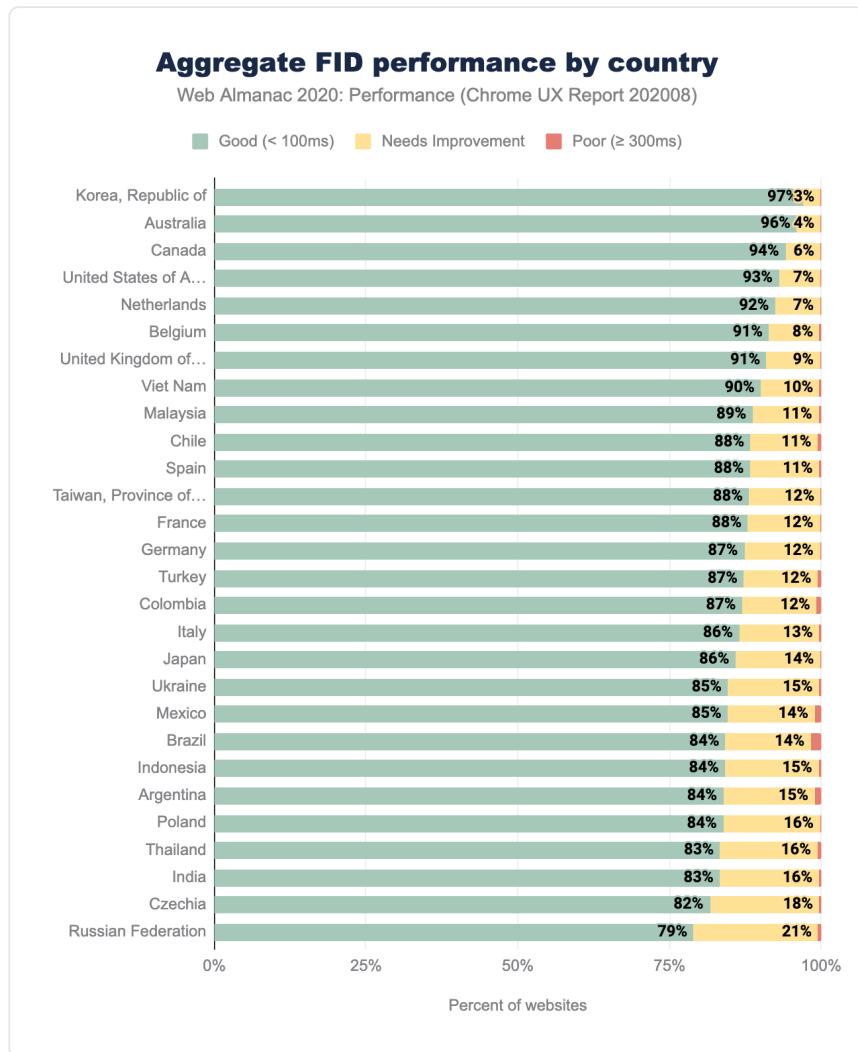


Figure 9.10. Aggregate FID performance split by country.

The geographic distribution of FID scoring confirms the findings in the aggregate device chart shared earlier on. At worst, 79% of websites have good FID, with an impressive 97% on the top position with the Republic of Korea leading again. Interestingly, some top contenders from the CLS and LCP ranking, such as Czechia, Poland, Ukraine and Russian Federation here fall to the bottom of the hierarchy.

Again, it isn't easy to speculate why that might be. Assuming FID correlates to JavaScript execution capabilities, countries where more capable devices are more expensive and treated as luxury items, might report lower FID ranking. Poland is a good example—with one of the highest iPhone prices compared to the US market, combined with relatively lower wages, a single salary isn't sufficient to purchase Apple's flagship product. To contrast, Australians with average salaries would be able to buy an iPhone with a weeks' worth of pay. Luckily, the percentage of websites with a low rating is mostly at 0, with a few exceptions of 1-2% readings, pointing towards a relatively speedy response to the interaction.

## FID by connection type

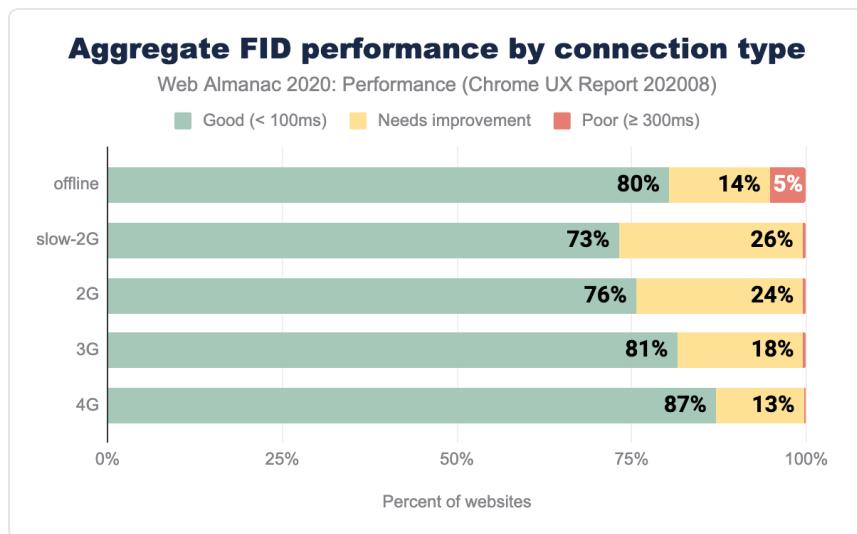


Figure 9.11. Aggregate FID performance split by connection type.

We can observe a direct correlation between network speed and fast FID, ranging from 73% on 2G to 87% on 4G networks. Faster networks will aid in speedier script download, which consequently speeds up the beginning of the parsing and fewer tasks blocking the main thread. It is optimistic to see such results, especially when the ratio of poorly rated site experiences doesn't exceed 5%.

## First Contentful Paint

First Contentful Paint (FCP) measures the first time at which the browser rendered any text, image, non-white canvas or SVG content. FCP is a good indicator of perceived speed as it portrays how long people wait to see the first signs of a site loading.

## FCP by device

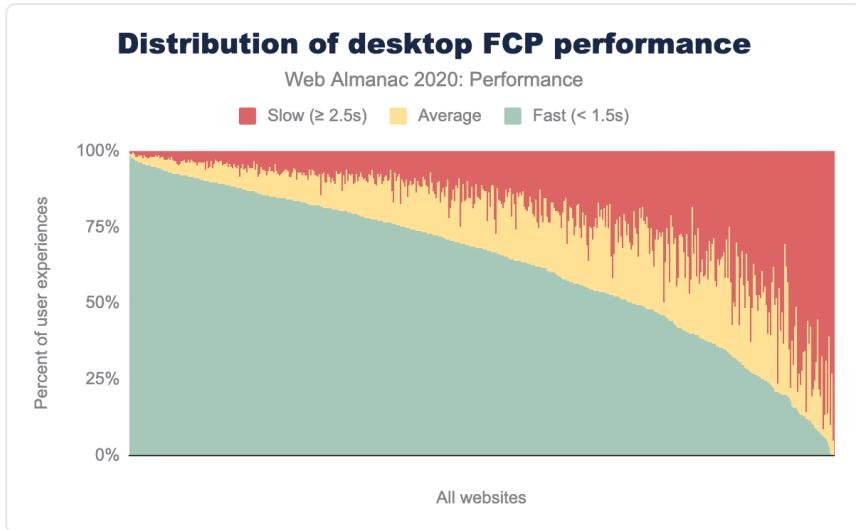


Figure 9.12. Distribution of websites labeled as having fast, average and slow FCP performance on desktop.

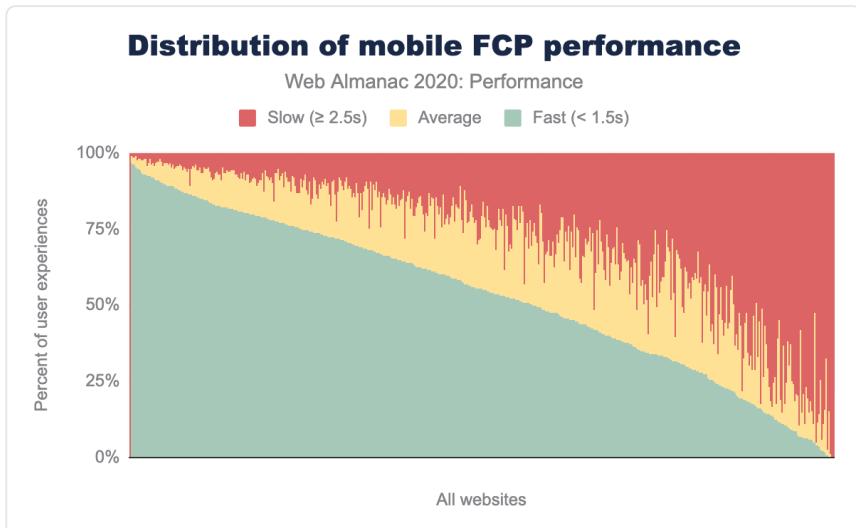
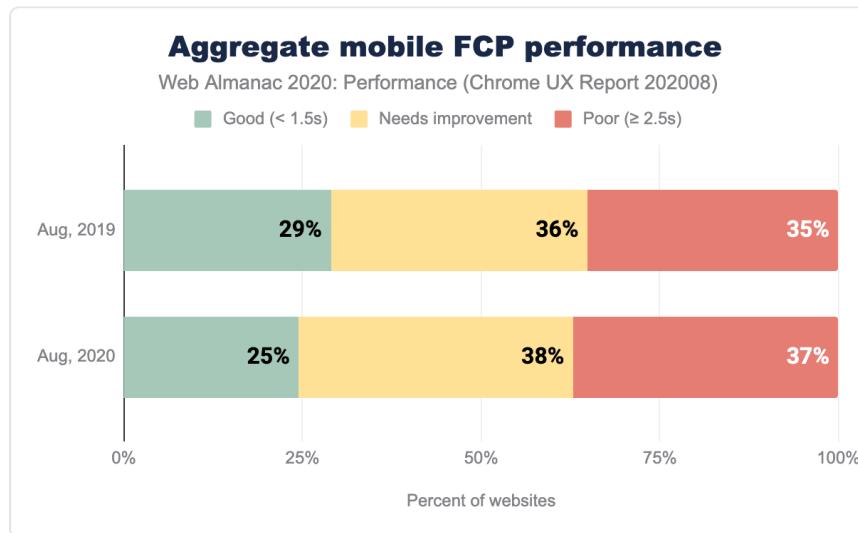


Figure 9.13. Distribution of websites labeled as having fast, average and slow FCP performance on mobile.

In the charts above, the FCP distributions are broken down by desktop and mobile. Comparing

to last year, there are noticeably less average FCP readings, while the percentage of fast and slow user experiences has risen no matter the device type. We can still observe the same trend, where mobile users will experience slower FCP more frequently than desktop users. Overall, users are more likely to have a good or poor experience, rather than a mediocre experience.



*Figure 9.14. A comparison of distribution of websites labeled as having good, needs improvement and poor FCP mobile performance between 2019 and 2020.*

Comparing FCP on mobile devices on a year-over-year basis, we observe fewer good experiences and more moderate and poor experiences. 75% of websites have sub-par FCP. We can speculate this high percentage of less than ideal FCP readings is a source of frustration and degraded user experience.

Numerous factors can delay paints, such as server latency (measured by a handful of metrics, such as Time to First Byte (TTFB) and RTT), blocking JavaScript requests, or inappropriate handling of custom fonts to name a few.

## FCP by geographic location

### Aggregate FCP performance by country

Web Almanac 2020: Performance (Chrome UX Report 202008)

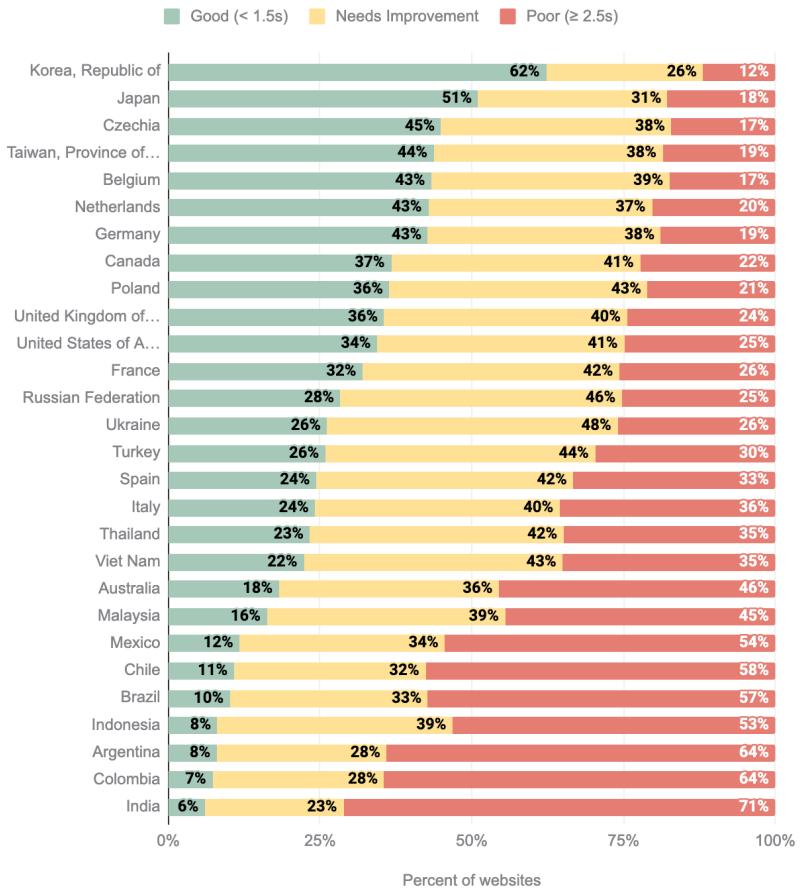


Figure 9.15. Aggregate FCP performance split by country.

Before we dig into the analysis, it is noteworthy to mention that in the 2019 Performance chapter, the thresholds for "good" and "poor" classification were different from 2020. In 2019, sites with FCP below 1s were considered good, while those with FCP above 3s were categorized as poor. In 2020, those ranges shifted to 1.5s for good and 2.5s for poor.

This change means that the distribution would shift towards more "good" and "poor" rated

websites. We can observe that trend compared to last year's results, as the percentage of good and poor websites rise. The top ten geographies with the highest rate of fast websites remain relatively unchanged from 2019, with the addition of Czechia and Belgium and the fall of the United States and the United Kingdom. The Republic of Korea leads with 62% of websites reporting fast FCP, nearly doubling since last year (which, again, is likely due to re-categorization of results). Other countries in the top of the ranking also double the number of good experiences.

While the mediocre ("needs improvement") percentage becomes smaller, the number of poorly performing FCP sites rises, which is especially pronounced at the bottom of the ranking with Latin American and South Asian regions.

Again, there are several reasons negatively affecting FCP, such as poor TTFB readings, but it is difficult to prove them without the necessary context. For example, if we were to analyze specific country performance, such as Australia, we surprisingly find it at the lower end. Australia has one of the highest global smartphone penetration levels, one of the fastest mobile networks and relatively high average income levels. We'd easily assume it should be higher up. However, taking into account slow fixed connections, latency and lack of iOS representation in CrUX, its positioning starts making more sense. With an example like this (touched only on the surface), we can see how difficult understanding context for each of the countries would be.

## FCP by connection type

### Aggregate FCP performance by connection type

Web Almanac 2020: Performance (Chrome UX Report 202008)

Good (< 1.5s)     Needs improvement     Poor (≥ 2.5s)

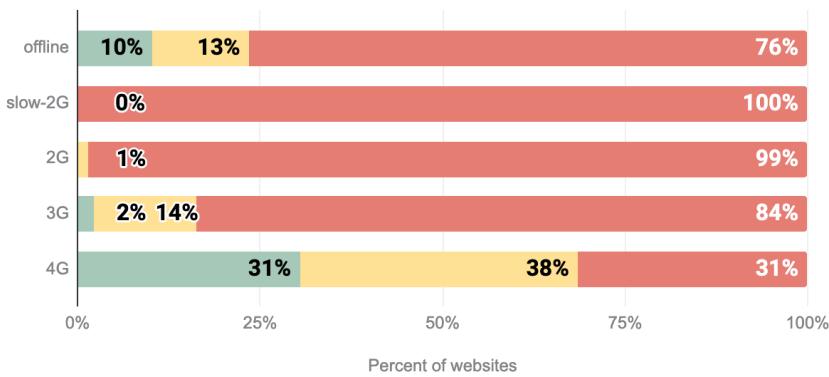


Figure 9.16. Aggregate FCP performance split by connection type.

Similarly to other metrics, FCP is affected by connection speeds. On 3G, only 2% of experiences

rate good, while on 4G, 31%. It is not an ideal state of FCP performance, but it has improved since 2019 in some areas, which again might be driven by the change in categorization of good and poor categorization. We see the same rise in the percentage of good websites and poor websites, narrowing the number of moderate ("needs improvement") site experiences.

This trend illustrates the furthering digital divide, where experiences on slower networks and potentially less capable devices are consistently worse. Improving FCP on slow connections directly correlates to enhancing TTFB, which we observe in Aggregate TTFB performance by connection type chart—poor TTFB = poor FCP.

The choice of hosting provider or CDN will have a cascading effect on speed. Making these decisions based on the fastest possible delivery will help in improving FCP and TTFB, especially on slower networks. FCP is also significantly affected by font load time, so ensuring text is visible while web fonts are downloaded is also a worthwhile strategy (especially where on slower connections these resources will be costly to fetch).

Looking at the "offline" statistics, we can deduce that a substantial number of FCP issues are also *not* correlated to the network type. We don't observe significant gains in this category, which we would if that statement was true. It appears would seem rendering is not so much delayed by fetching JavaScript, but it is affected by parsing and execution.

## Time to First Byte

Time to First Byte (TTFB) is the time taken from the initial HTML request being made until the first byte arrives back to the browser. Issues with swiftly processing requests can quickly cascade into affecting other performance metrics as they will delay not only paints but also any resource fetching.

## TTFB by device

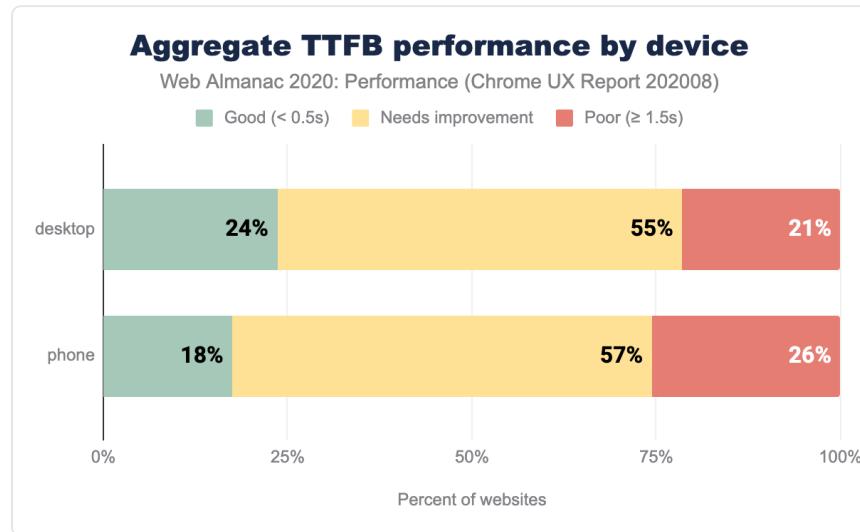


Figure 9.17. Aggregate TTFB performance split by device type.

On desktop, 76% of websites have a "not good" TTFB, while on mobile, that percentage rises to 83%. We might assume that the data portrays how TTFB is often an overlooked metric when it is assumed that most performance measurements and work is concentrated within front-end and visual rendering, not asset delivery and server-side work. High TTFB will have a direct, negative impact on a plethora of other performance signals, which is an area that still needs addressing.

## TTFB by geographic location

### Aggregate TTFB performance by country

Web Almanac 2020: Performance (Chrome UX Report 202008)

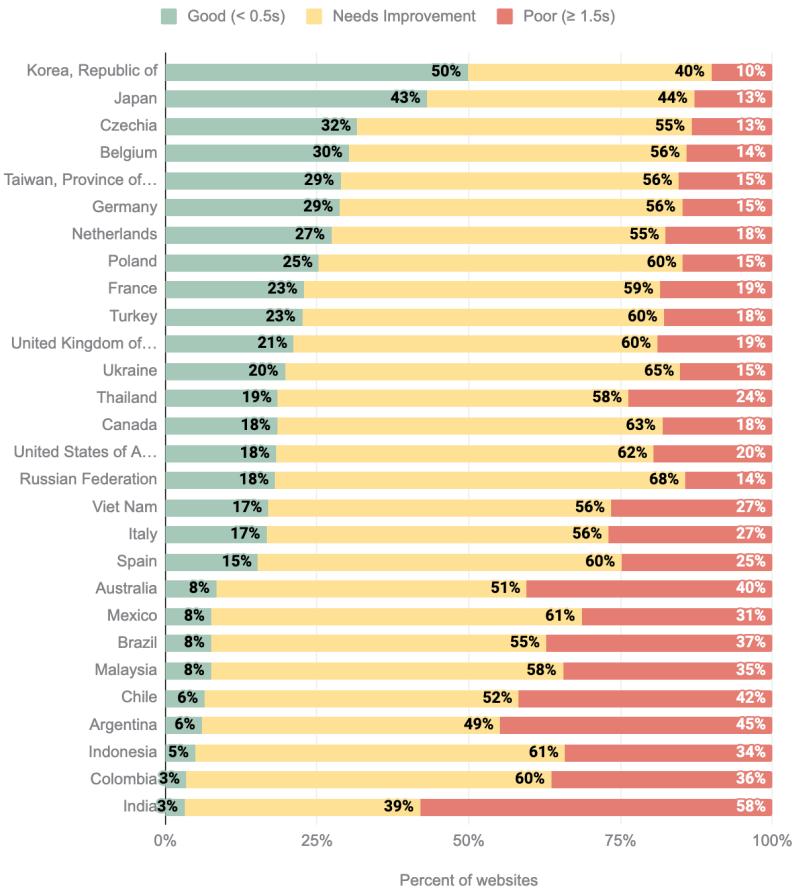


Figure 9.18. Aggregate TTFB performance split by country.

Likening this years' TTFB geo readings to 2019 results again points to more fast websites, but similarly to FCP, the thresholds have changed. Previously, we considered TTFB below 200ms fast, and above 1000ms slow. In 2020, TTFB below 500ms is good and above 1500ms poor. Such generous changes in categorization can explain that we observe significant changes, such as a 36% rise in good website experiences in The Republic of Korea or 22% rise in Taiwan.

Overall, we still observe similar regions, such as Asia-Pacific and selected European locales leading.

## TTFB by connection type

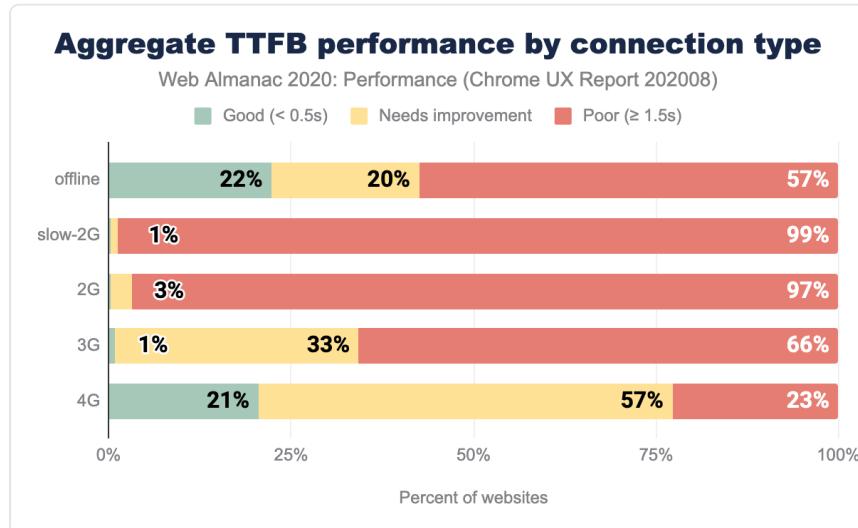


Figure 9.19. Aggregate TTFB performance split by connection type.

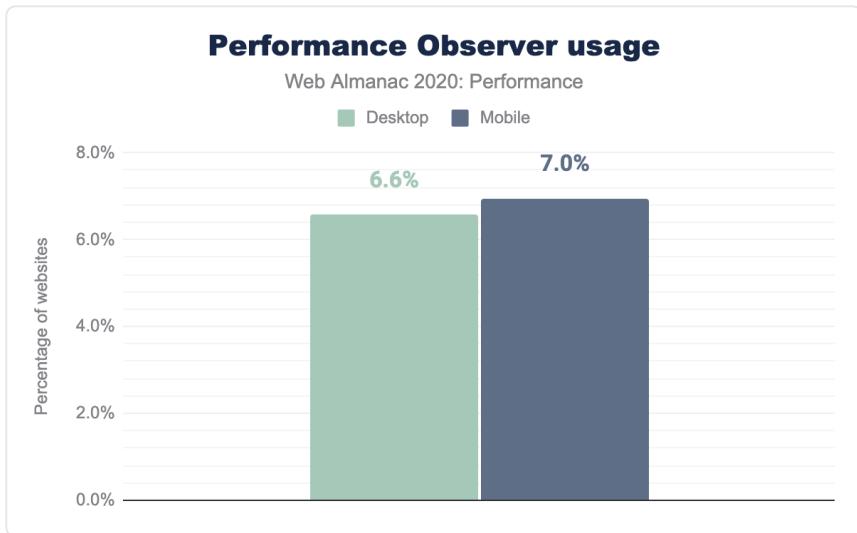
TTFB is affected by network latency and connection type. The higher the latency and the slower the connection, the worse TTFB measurements, as we can observe above. Even on mobile connections considered as fast (4G), only 21% of websites have a fast TTFB. There are nearly no sites categorized as quick below 4G speeds.

Looking at the mobile speeds worldwide for December 2018–November 2019, we can see that globally, mobile connections aren't high-speed. Those network speeds and technology standards for cellular networks (such as 5G) are not evenly distributed and affect TTFB. As an example, see this map of networks in Nigeria—most of the country area has 2G and 3G coverage, with little 4G range.

What's surprising is the relatively the same number of good TTFB results between offline and 4G origins. With service workers, we could expect some of the TTFB issues to be mitigated, but that trend is not reflected in the chart above.

## Performance Observer usage

There are dozens of different user-centric metrics that can be used to assess websites and applications. However, sometimes the predefined metrics don't quite fit our specific scenarios and needs. The `PerformanceObserver` API allows us to obtain custom metric data obtained with User Timing API, Long Task API, [Event Timing API](Event Timing API) and a handful of other low-level APIs. For example, with their help, we could record the timing transitions between pages or quantify server-side-rendered (SSR) application hydration.



*Figure 9.20. Performance Observer usage by device type.*

The chart above showcases that Performance Observer is used by 6-7% of tracked sites, depending on device type. Those websites will be leveraging the low-level APIs to create custom metrics, and the `PerformanceObserver` API to collate them, and then potentially use it with other performance reporting tooling. Such adoption rates might indicate the tendency to lean on predefined metrics (for example, coming from Lighthouse), but also are impressive for a relatively niche API.

## Conclusion

User experience is not only a spectrum but also depends on a wide variety of factors. To attempt understanding the state of performance without excluding the sub-par, underprivileged experiences, we must approach it intersectionally. Each website visit tells a story. Our personal and country-level socioeconomic status dictates the type of device and

internet provider we can afford. The geopositioning of where we live affects latency (we Australians feel this pain regularly), and the economy dictates available cellular network coverage. What websites do we visit? What do we visit them for? Context is critical to not only analyzing data but also developing necessary empathy and care in building accessible, fast experiences for all.

On the surface, we have seen optimistic signals about the new Core Web Vitals performance metrics. At least half of the experiences are good across both desktop and mobile devices, if we don't narrow down to consistently poor experiences on slower networks for Largest Contentful Paint. While the newer metrics might suggest that there's an ongoing uptake in addressing performance issues, the lack of significant improvements in First Contentful Paint and Time to First Byte is sobering. Here the same network types are most disadvantaged as with Largest Contentful Paint, as well as fast connections and desktop devices. The Performance Score also portrays a decline in speed (or perhaps, a more accurate portrayal than what we measured in the past).

What the data shows us, is that we must keep investing in improving performance for scenarios (such as slower connectivity) that we often don't experience due to multiple aspects of our privilege (middle to high-income countries, high pay and new, capable devices). It also highlights that there's still plenty of work to be done in the areas of speeding up initial paints (LCP and FCP) and asset delivery (TTFB). Often, performance feels like an inherently front-end issue, while numerous significant improvements can be achieved on the back-end and through appropriate infrastructure choices. Again, user experience is a spectrum that depends on a variety of factors, and we need to treat it holistically.

New metrics bring new lenses to analyze user experience through, but we must not forget existing signals. Let's focus on moving the needle in the areas that need the most improvement and will result in positive shifts in experience for most underserved. Fast and accessible internet is a human right.

---

## Author

---



Karolina Szczur

 @fox  thefoxis

Karolina is a Product Design Lead at Calibre<sup>24</sup>, working on creating the most comprehensive speed monitoring platform. She curates Performance Newsletter<sup>25</sup>, your source of performance news and resources. Karolina also frequently writes<sup>26</sup> about how performance affects user experience.

---

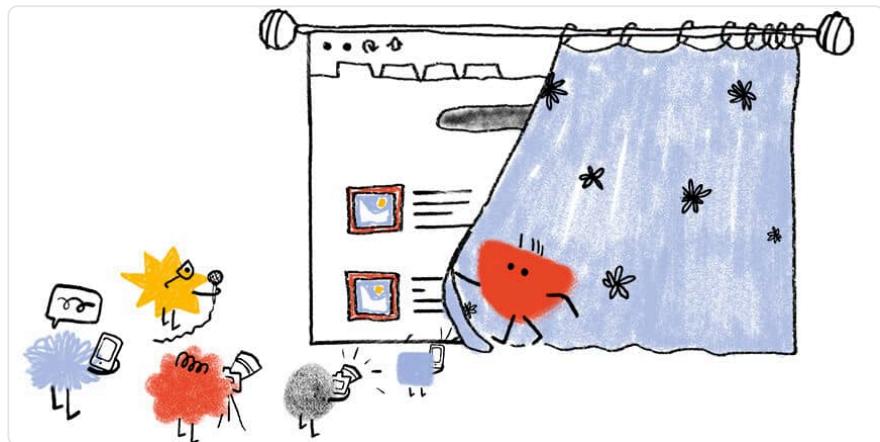
24. <https://calibreapp.com/>

25. <https://perf.email/>

26. <https://calibreapp.com/blog/category/web-platform>

# Part II Chapter 10

# Privacy [UNEDITED]



Written by Yana Dimova

Reviewed by Laurent Devernay

Analyzed by Yana Dimova and Max Ostapenko

## Introduction

This chapter of the Web Almanac gives an overview of the current state of privacy on the web. This topic has been increasing in popularity recently and has raised awareness on the users' side. The need for guidelines has been met with various regulations (such as GDPR in Europe, LGPD in Brazil, CCPA in California...). These aim to increase the accountability of data processors and their transparency towards users. In this chapter, we discuss the prevalence of online tracking with different techniques and the adoption rate of cookie consent banners and privacy policies by websites.

## Online Tracking

Third-party trackers collect user data to build up profiles of the user's behavior to be monetized for advertising purposes. This raises privacy concerns with users on the web, which resulted in the emergence of various tracking protections. However, as we will see in this section, online

tracking is still widely used. Not only does it have a negative impact on privacy, online tracking has a huge impact on the environment and avoiding it can lead to better performance. We examine the prominence of the most common types of third-party tracking, namely by means of third-party cookies and the use of fingerprinting. Online tracking is not limited to just these two techniques, new ones keep arising to circumvent existing countermeasures.

## Third-party trackers

We use WhoTracksMe's tracker list to determine the percentage of websites that issue a request to a tracker. As in the following figure, we find that on roughly 93% of websites, at least one tracker is present.

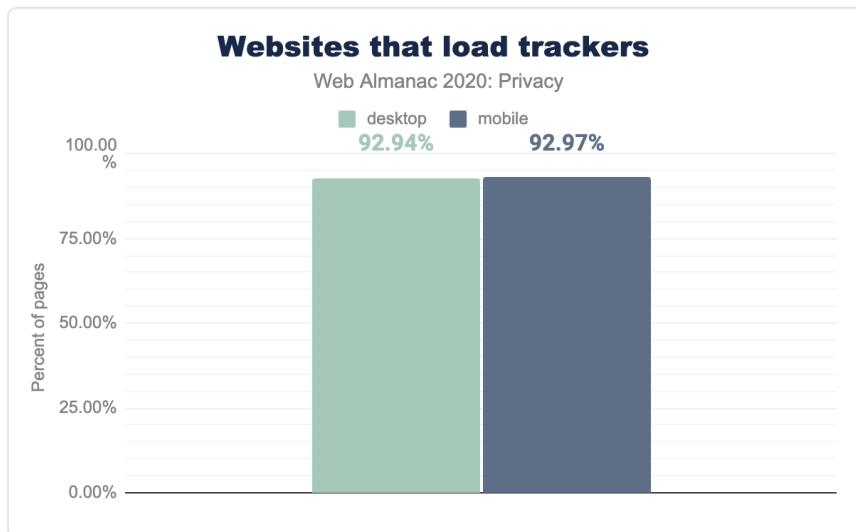


Figure 10.1. Websites including at least one tracker

We examine the largest (most widely used) trackers and plot the prevalence of the 10 most popular ones.

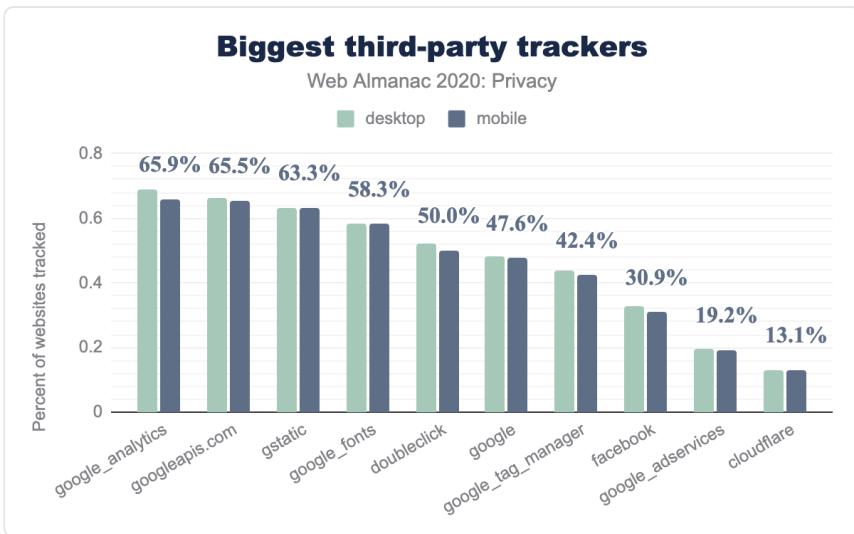


Figure 10.2. Top 10 Trackers

The largest player on the online tracking market is definitely Google, with eight of its tracking domains present in the top 10 trackers and prevalent on at least 70% of websites. Following are Facebook and Cloudflare.

WhoTracksMe's tracker list also defines categories that the trackers belong to. The following figure shows the distribution of the different categories for the 100 largest trackers.

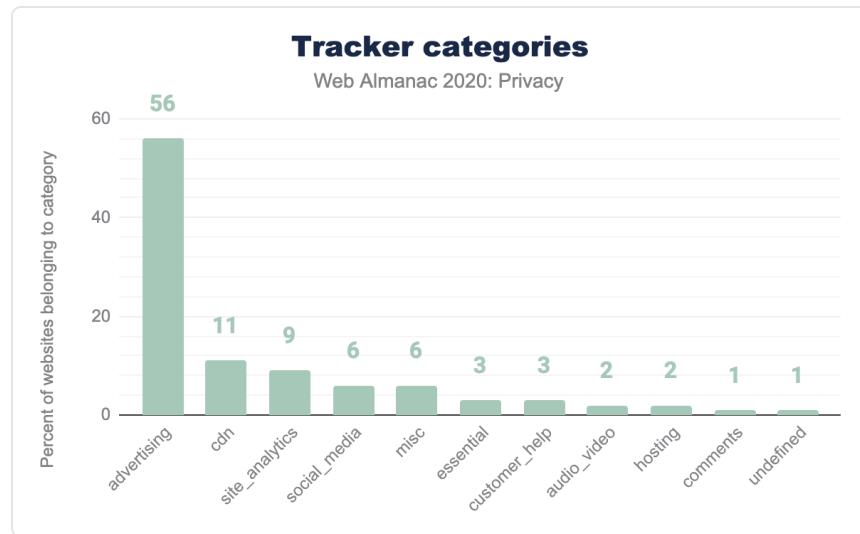


Figure 10.3. Categories of the 100 most popular trackers

Nearly 60% of the most popular trackers are advertising-related. This could be due to the profitability of the online advertising market.

## Cookies

We looked into the most popular cookies being set on websites in HTTP's response header, according to their name and domain.

<b>Domain</b>	<b>Cookie Name</b>	<b>Websites present on</b>
doubleclick.net	test_cookie	24%
facebook.com	fr	10%
youtube.com	VISITOR_INFO1_LIVE	10%
youtube.com	YSC	10%
doubleclick.net	IDE	9%
doubleclick.net	unknown	9%
youtube.com	GPS	9%
doubleclick.net	unknown	8%
google.com	NID	6%
doubleclick.net	unknown	6%

Figure 10.4. Top cookies on desktop sites

<b>Domain</b>	<b>Cookie Name</b>	<b>Websites present on</b>
doubleclick.net	test_cookie	32%
doubleclick.net	IDE	21%
facebook.com	fr	10%
youtube.com	VISITOR_INFO1_LIVE	10%
youtube.com	YSC	10%
google.com	NID	10%
youtube.com	GPS	8%
doubleclick.net	DSID	7%
yandex.ru	yandexuid	6%
yandex.ru	i	6%

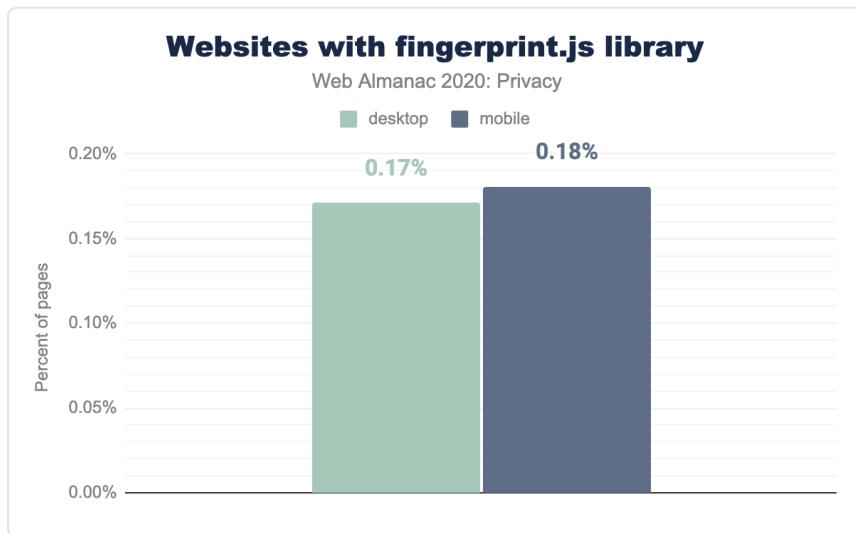
Figure 10.5. Top cookies on mobile sites

As found previously, Google's tracking domain 'doubleclick.net' sets cookies on roughly one fourth of websites on a mobile client and one third of all websites on a desktop client. Again, nine out of the ten most popular cookies on desktop client and 7 on mobile are set by a Google

domain. This is a lower bound for the amount of websites the cookie is set on, since we are only counting cookies set via an HTTP header, a large amount of tracking cookies is set by using third-party scripts.

## Fingerprinting

Another widely-used tracking technique is fingerprinting, which consists of collecting different kinds of information about the user with the goal of building a unique “fingerprint” for them. Different types of fingerprinting are used on the web by trackers. Browser fingerprinting uses characteristics specific to the browser of the user, relying on the fact that the chance of another user having the exact same browser information is fairly small. In our crawl, we examined the presence of the FingerprintJS library, which provides browser fingerprinting as a service.



*Figure 10.6. Websites using FingerprintJS*

Although the library is present on a small percentage of websites, the main issue with fingerprinting is its persistent nature. Furthermore, FingerprintJS is not the only attempt at fingerprinting. Other libraries, tools and native code can also serve this purpose.

## Consent Management Platforms

Cookie consent banners have become common now, increase transparency towards cookies and often allowing users to specify their cookie choices. While a lot of websites opt for using their own implementation of cookie banners, third-party solutions called 'Consent

Management Platforms' have recently emerged. The platforms provide an easy way for websites to collect user's consent for different types of cookies. We see that 4.4% of websites use some consent management platform to provide cookie choices on desktop clients, and 4.0% on mobile clients.

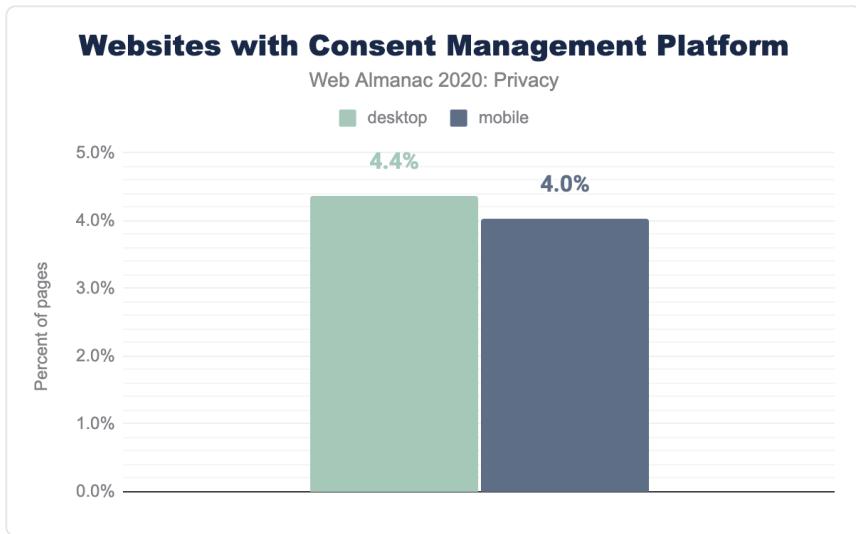


Figure 10.7. Websites using a consent management platform

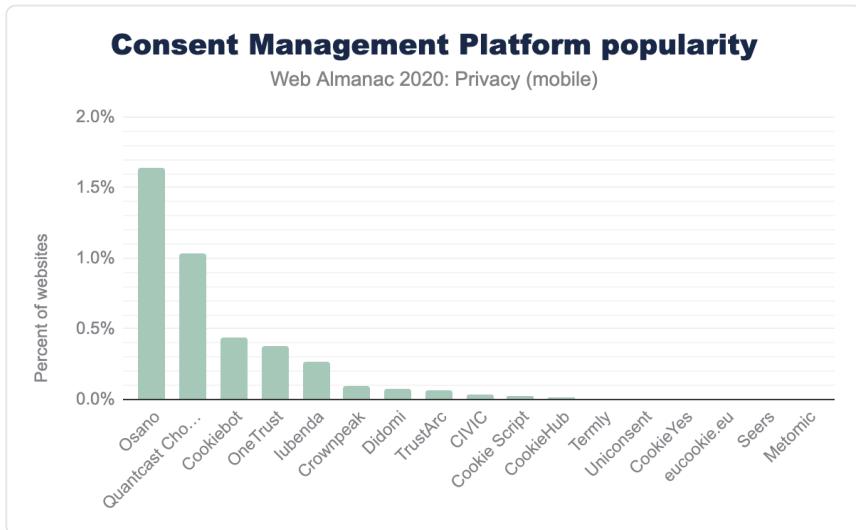
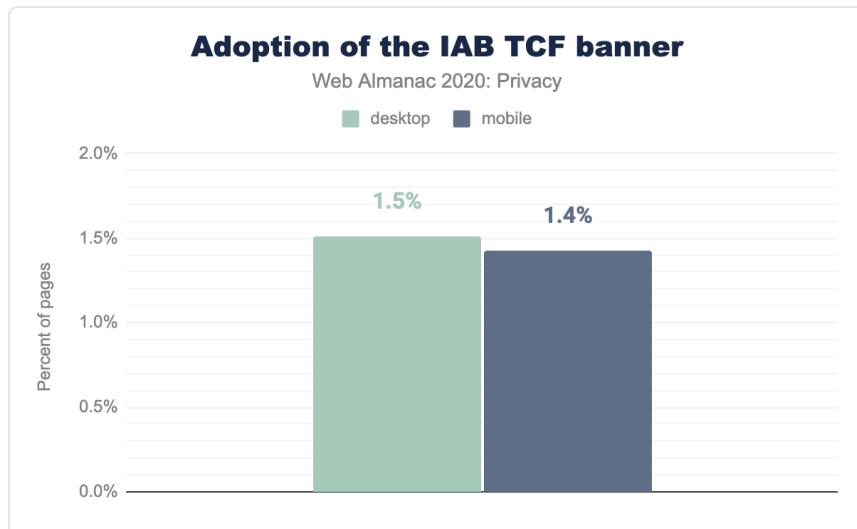


Figure 10.8. Popularity of consent management platform

When looking at the popularity of the different consent management solutions, we can see that Osano and Quantcast choice are the leading platforms.

## IAB Europe's Transparency Consent Framework

IAB Europe, the European association for digital marketing and advertising, proposed a Transparency Consent Framework (TCF) as a GDPR-compliant solution to obtain users' consent about their digital advertising preferences. The implementation provides an industry standard for communication between publishers and advertisers about consumer consent.



*Figure 10.9. Adoption rate of TCF banner*

While our results show that the TCF banner is not yet the 'industry standard', it is a step in the right direction. Considering the main target group of IAB Europe is in fact European publishers, having an adoption rate on 1.5% of websites on desktop client and 1.4% on mobile is not too bad.

## Privacy Policies

Privacy policies are widely used by websites to meet legal obligations and increase transparency towards users about data collection practices. In our crawl, we searched for keywords indicating the presence of a privacy policy text on each visited website.

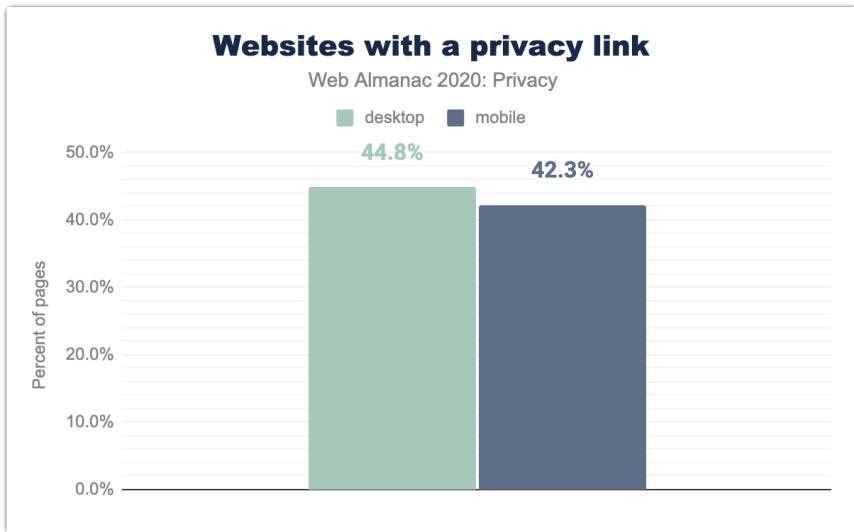


Figure 10.10. Websites that have a privacy policy

The results show that almost half of the websites in the dataset have included a privacy policy, which is positive. However, studies have shown that the majority of internet users do not bother reading privacy policies and when they do, they lack understanding due to the length and complexity of most privacy policy texts.

## Conclusion

This chapter gives an overview of the current state of privacy on the web. Third-party tracking remains prominent on both desktop and mobile clients, with Google tracking the largest percentage of websites. Consent Management Platforms are used on a small percentage of websites, however a lot of websites implement their own cookie consent banners. Lastly, roughly half of the websites include a privacy policy, which benefits greatly transparency towards users about data processing practices. This is undoubtedly a step forward but there is a lot to be done. Often, privacy policies are hard to read and understand and cookie consent banners manipulate users into consent. For the web to truly respect users, privacy has to be a part of conception, not an afterthought. Regulation is a good thing and it is reassuring to see an increase in privacy regulation worldwide but, more than that and financial sanctions, privacy by design should be the norm, rather than deploying policies and tools in order to avoid fines.

## Author

---



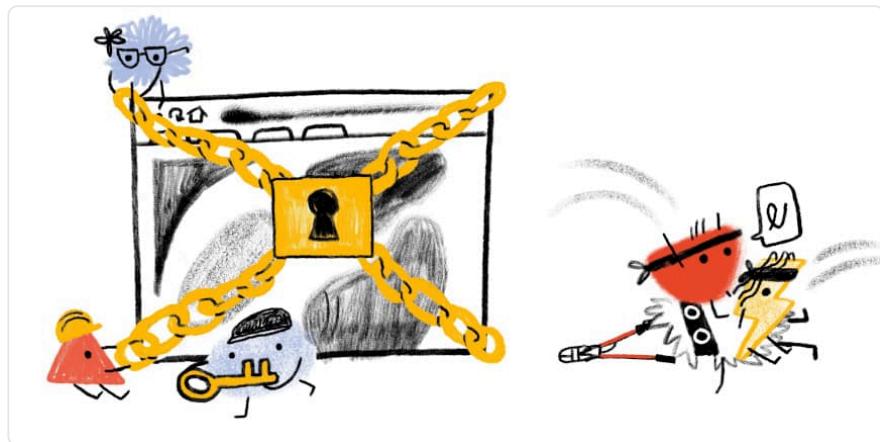
Yana Dimova

ydimova

---

# Part II Chapter 11

# Security [UNEDITED]



Written by Nurullah Demir and Tom Van Goethem

Reviewed by Caleb Queern, Barry Pollard, and Edmond W. W. Chan

Analyzed by Akshar Agarwal and Tom Van Goethem

## Authors



Nurullah Demir

Twitter: @nrllah GitHub: nrllh Website: <https://internet-sicherheit.de>



Tom Van Goethem

Twitter: @tomvangoethem GitHub: tomvangoethem



## Part II Chapter 12

# Mobile Web [UNEDITED]



**Written by Shubhie Panicker and Michael DiBlasio**

Reviewed by Jeremy Wagner, David Fox, and Cheney Tsai

Analyzed by David Fox

### Authors



Shubhie Panicker

 @shubhie  spanicker



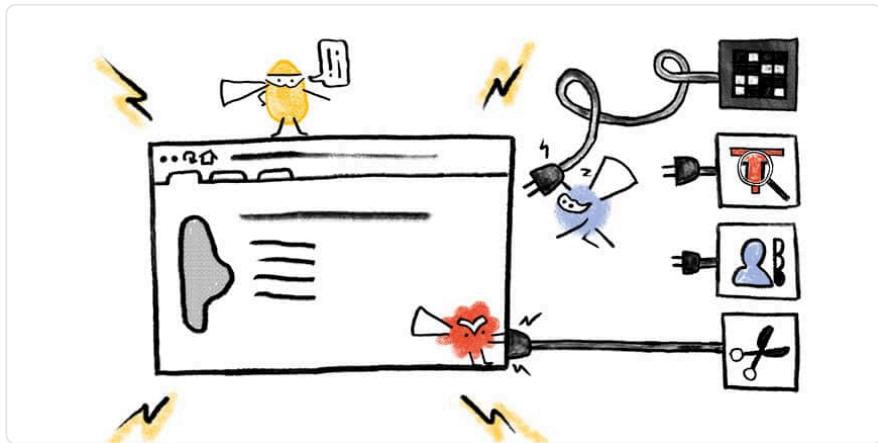
Michael DiBlasio

 mdiblasio



# Part II Chapter 13

# Capabilities



Written by Christian Liebel

Reviewed by Thomas Steiner

Analyzed by Thomas Steiner

## Introduction

Progressive Web Apps (PWA) are a cross-platform application model based on web technology. With the help of Service Workers, these applications run even when the user is offline. The Web App Manifest allows users to add a PWA to their home screen or program list. When opened from there, a PWA appears as a native application. However, PWAs can only use the functions and capabilities that are exposed through web platform APIs. Arbitrary native interfaces cannot be called, leaving a gap between native applications and web apps.

The Capabilities Project, informally also known as Project Fugu, is a cross-company effort by Google, Microsoft, and Intel to bridge the gap between web and native. This is important to keep the web relevant as a platform. To do so, the Chromium contributors implement new APIs exposing capabilities of the operating system to the web, while maintaining user security, privacy, and trust. These capabilities include, but are not limited to:

- File System Access API for accessing files on the local file system
- File Handling API for registering as a handler for certain file extensions

- Async Clipboard API to access the user's clipboard
- Web Share API for sharing files with other applications
- Contact Picker API to access contacts from the user's address book
- Shape Detection API for efficient detection of faces or barcodes in images
- Web NFC, Web Serial, Web USB, Web Bluetooth, and other APIs (for the entire list, see the Fugu API Tracker)

Anyone can propose a new capability by creating a ticket in the Chromium bug tracker. The Chromium contributors examine the proposals and discuss all APIs with other developers and browser vendors through the appropriate standards bodies. Meanwhile, the Fugu team implements the API in Chromium, where it is initially implemented behind a flag. Later in the process, the API is made available to a limited audience via an origin trial. During this phase, developers can sign up for a token to test the API on a specific origin. If the API turns out to be robust enough, the API ships in Chromium and, if the vendors decide so, other browsers. The Capability Status site shows where the different Capability APIs are in the process.

Project Fugu, the codename of the Capabilities Project, is named after a Japanese dish: correctly prepared, the meat of the blowfish is a special taste experience. If prepared incorrectly, however, it can be fatal. The powerful APIs of Project Fugu are extremely exciting for developers. However, they can affect the security and privacy of the user. Therefore, the Fugu team pays special attention to these issues. For instance, new interfaces require the website to be sent over a secure connection (HTTPS). Some of them require a user gesture, such as a click or key press, to prevent fraud. Other capabilities require explicit permission by the user. Developers can use all APIs as a progressive enhancement: by feature detecting the APIs, applications won't break in browsers lacking support for those capabilities. In browsers that support them, users can get a better experience. This way, web apps progressively enhance according to the user's particular browser.

This chapter gives an overview of various modern web APIs, and the state of web capabilities in 2020 based on usage data by the HTTP Archive and Chrome Platform Status. Since some interfaces are brand-new, their (relative) usage is very low. So, unlike most chapters, HTTP Archive usage stats will be presented as the absolute numbers of pages rather than relative percentages. Due to technical limitations, the HTTP Archive only has data available for APIs that require neither permission, nor a user gesture. Where no data is available, the percentage of page loads in Google Chrome according to Chrome Platform Status will be shown instead. Even if some figures are so small that the statistics are not necessarily meaningful, in many cases trends can still be read from the data. Also, these stats can be used as a baseline for future annual editions of this chapter, looking back to see how much the APIs have matured and improved their adoption. Unless otherwise noted, the APIs are only available in Chromium-based browsers, and their specifications are in the early stages of standardization.

## Async Clipboard API

With the help of the `document.execCommand()` method, websites could already access the user's clipboard. However, this approach is somewhat restricted, as the API is synchronous (making it difficult to process clipboard items), and it can only interact with selected text in the DOM. This is where the Async Clipboard API (W3C Working Draft) comes in. This new API is not only asynchronous, meaning it doesn't block the page for large chunks of data or waiting for a permission to be granted, but it also allows for images to be copied to or pasted from the clipboard in supported browsers such as Chrome, Edge, and Safari.

### Read Access

The Async Clipboard API provides two methods for reading content from the clipboard: a shorthand method for plain text, called `navigator.clipboard.readText()`, and a method for arbitrary data, called `navigator.clipboard.read()`. Currently, most browsers only support HTML content and PNG images as additional data formats. As the clipboard may contain sensitive data, reading from it requires the user's consent.

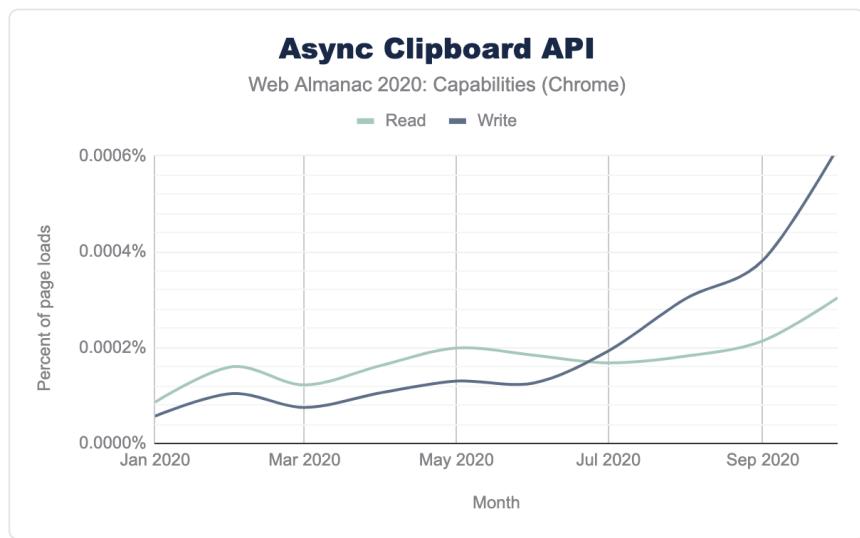


Figure 13.1. Percentage of page loads in Chrome using Async Clipboard API.  
(Sources: Async Clipboard Read, Async Clipboard Write)

The Async Clipboard API is comparatively new, so its usage is currently rather low. In March 2020, Safari added support for the Async Clipboard API in Safari 13.1. Over the course of 2020, the usage of the `read()` API was growing. In October 2020, the API was called during 0.0003% of all page loads in Google Chrome.

## Write Access

Apart from reading operations, the Async Clipboard API also offers two methods for writing content to the clipboard. Again, there's a shorthand method for plain text, called `navigator.clipboard.writeText()`, and one for arbitrary data called `navigator.clipboard.write()`. In Chromium-based browsers, writing to the clipboard while the tab is active does not require permission. Trying to write to the clipboard when the website is in the background does, however. As this method requires a user gesture and permission first, it's not covered by the HTTP Archive metrics. In contrast to the `read()` method, the `write()` method shows an exponential growth in usage, being part of 0.0006% of all page loads in October 2020.

The Raw Clipboard Access API, another Fugu capability, might even further enhance the Async Clipboard API by allowing arbitrary data to be copied from or pasted to the clipboard.

## StorageManager API

Web browsers allow users to store data on the user's system in different ways, such as Cookies, the Indexed Database (IndexedDB), the Service Worker's Cache Storage, or Web Storage (LocalStorage, Session Storage). In modern browsers, developers can easily store hundreds of megabytes and even more, depending on the browser. When browsers run out of space, they can clear data until the system is no longer over the limit, which can lead to data loss.

Thanks to the StorageManager API, which is part of the WHATWG Storage Living Standard, browsers no longer behave like a black box in that regard. This API allows developers to estimate the remaining space available and opt-in to persistent storage, meaning that the browser will not clear a website's data when disk space is low. Therefore, the API introduces a `new StorageManager` interface on the `navigator` object, currently available on Chrome, Edge, and Firefox.

### Estimate the available storage

Developers can estimate the available storage by calling `navigator.storage.estimate()`. This method returns a promise resolving to an object with two properties: `usage` shows the number of bytes used by the application and `quota` contains the maximum number of bytes available.

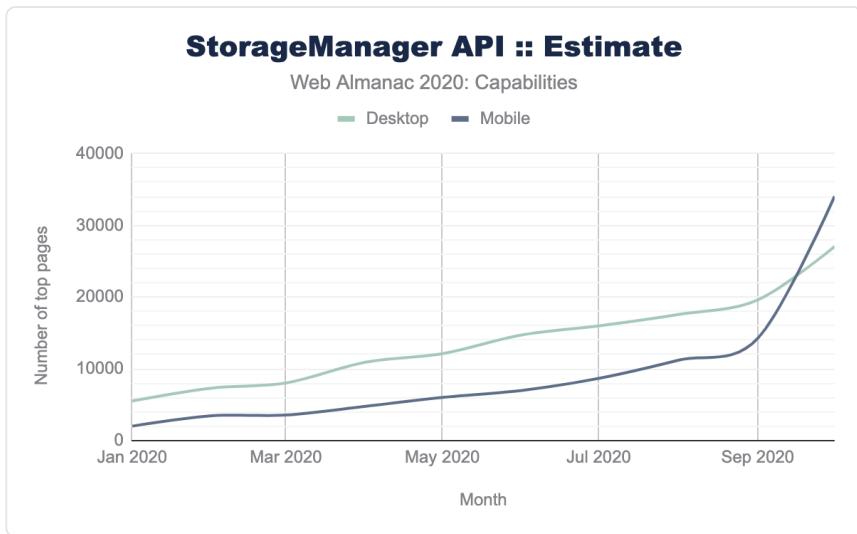


Figure 13.2. Number of pages using the estimate method of the StorageManager API.

The Storage Manager API is supported in Chrome since 2016, Firefox since 2017, and the new Chromium-based Edge. HTTP Archive data shows that the API is used on 27,056 desktop pages (0.49%) and 34,042 mobile pages (0.48%). Over the course of 2020, the usage of the Storage Manager API kept growing. This also makes this interface the most commonly used API in this chapter.

## Opt-in to persistent storage

There are two categories of web storage: "Best Effort" and "Persistent", with the first being the default. When a device is low on storage, the browser automatically tries to free up best effort storage. For instance, Firefox and Chromium-based browsers evict storage from the least recently used origins. This, however, poses a risk of losing critical data. To prevent eviction, developers can opt for persistent storage. In this case, the browser will not clear the storage, even when space is low. Users can still choose to clear up the storage manually. To opt for persistent storage, developers need to call the `navigator.storage.persist()` method. Depending on the browser and site engagement, a permission prompt may show, or the request will automatically be accepted or denied.

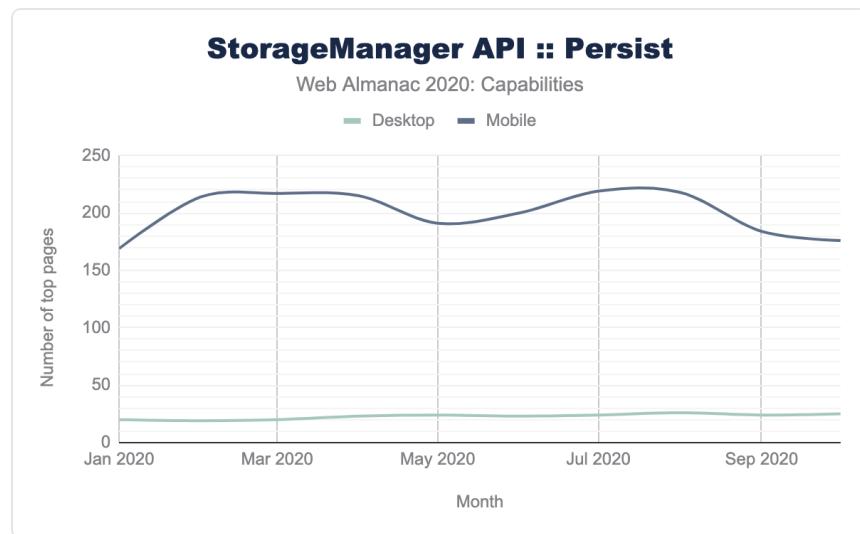


Figure 13.3. Number of pages using the persist method of the StorageManager API.

The `persist()` API is less often called than the `estimate()` method. Only 176 mobile pages make use of this API, compared to 25 desktop websites. While usage on the desktop seems to remain at this low level, there is no clear trend on mobile devices.

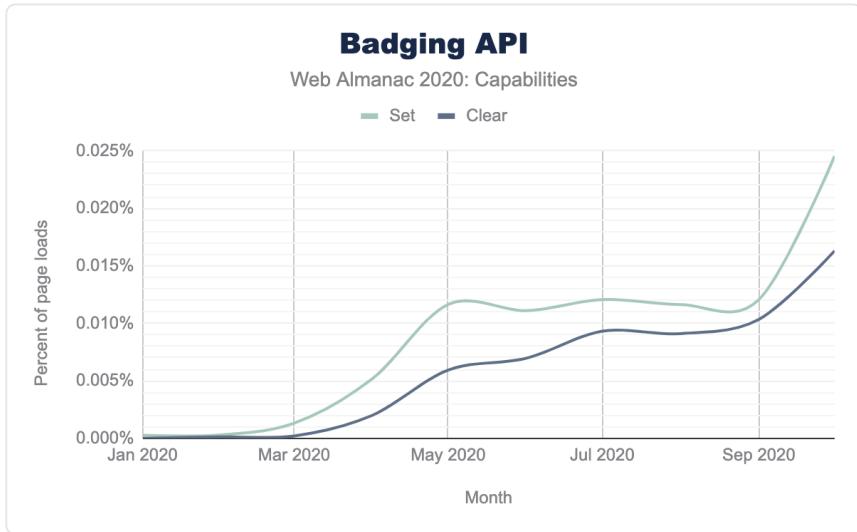
## New Notification APIs

With the help of the Push and Notifications APIs, web applications have long been able to receive push messages and display notification banners. However, some parts were missing. Until now, push messages had to be sent via the server; they could not be scheduled offline. Also, web applications installed to the system could not show badges on their icon. The Badging and Notification Triggers APIs enable both scenarios.

### Badging API

On several platforms, it's common for applications to present a badge on the application's icon indicating the amount of open actions. For instance, the badge could show the number of unread emails, notifications, or to-do items to complete. The Badging API (W3C Unofficial Draft) allows installed web applications to show such a badge on its icon. By calling `navigator.setAppBadge()`, developers can set the badge. This method takes a number to be shown on the application's badge. The browser then takes care of displaying the closest possible representation on the user's device. If no number is specified, a generic badge will be shown (e.g., a white dot on macOS). Calling `navigator.clearAppBadge()` removes the badge.

again. The Badging API is a great choice for email clients, social media apps, or messengers. The Twitter PWA makes use of the Badging API to show the number of unread notifications on the application's badge.



*Figure 13.4. Percentage of page loads in Chrome using Badging API.  
(Sources: Badge Set, Badge Clear)*

In April 2020, Google Chrome 81 shipped the new Badging API, followed by Microsoft Edge 84 in July. After Chrome shipped the API, the usage numbers shot up. In October 2020, on 0.025% of all page loads in Google Chrome, the `setAppBadge()` method is called. The `clearAppBadge()` method is less often called, during around 0.016% of page loads.

## Notification Triggers API

The Push API requires the user to be online to receive a notification. Some applications, such as games, reminder or to-do apps, calendars, or alarm clocks, could also determine the target date for a notification locally and schedule it. To support this feature, the Chrome team is experimenting with a new API called Notification Triggers (Explainer, not on a standards track yet). This API adds a new property called `showTrigger` to the `options` map that can be passed to the `showNotification()` method on the Service Worker's registration. The API is designed to allow for different kinds of triggers in the future, albeit for now, only time-based triggers are implemented. For scheduling a notification based on a certain date and time, developers can create a new instance of a `TimestampTrigger` and pass the target timestamp to it:

```
registration.showNotification('Title', {
  body: 'Message',
  showTrigger: new TimestampTrigger(timestamp),
});
```

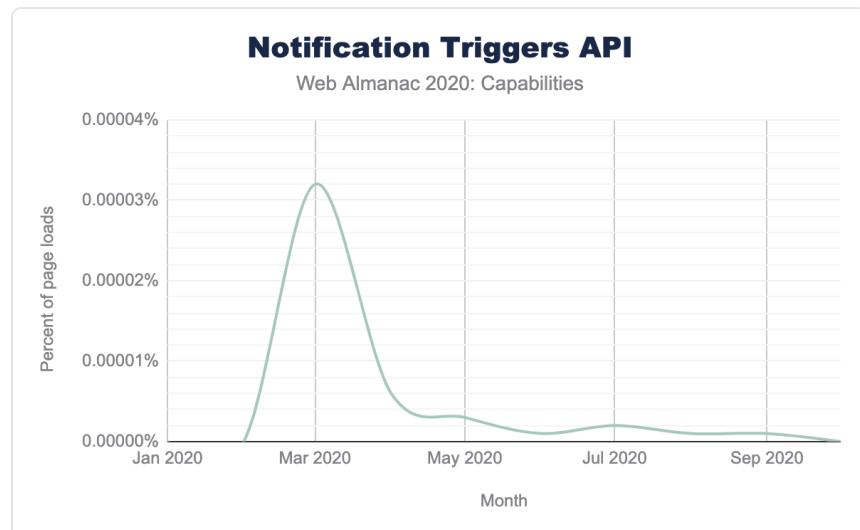


Figure 13.5. Percentage of page loads in Chrome using Notification Triggers API.  
(Source: Notification Triggers)

The Fugu team first experimented with Notification Triggers in an origin trial from Chrome 80 to 83, pausing development afterwards due to the lack of feedback by developers. Starting from Chrome 86 released in October 2020, the API has entered the origin trial phase again. This also explains the usage data of Notification Triggers API that peaked at being called on 0.000032% of page loads in Chrome during the first origin trial at around March 2020.

## Screen Wake Lock API

To save energy, mobile devices darken the screen backlight and eventually turn off the device's display, which makes sense in most cases. However, there are scenarios where the user may want the application to explicitly keep the display awake, for instance, when reading a recipe while cooking or watching a presentation. The Screen Wake Lock API (W3C Working Draft) solves this problem by providing a mechanism to keep the screen on.

The `navigator.wakeLock.request()` method creates a wake lock. This method takes a

`WakeLockType` parameter. In the future, the Wake Lock API could provide other lock types, such as turning the screen off, but keeping the CPU on. For now, the API only supports screen locks, so there is just one optional argument with the default value of 'screen'. The method returns a promise that resolves to a `WakeLockSentinel` object. Developers need to store this reference to call its `release()` method and release the screen wake lock later on. The browser will automatically release the lock when the tab is inactive, or the user minimizes the window. Also, the browser may deny a request and reject the promise, for example due to low battery.

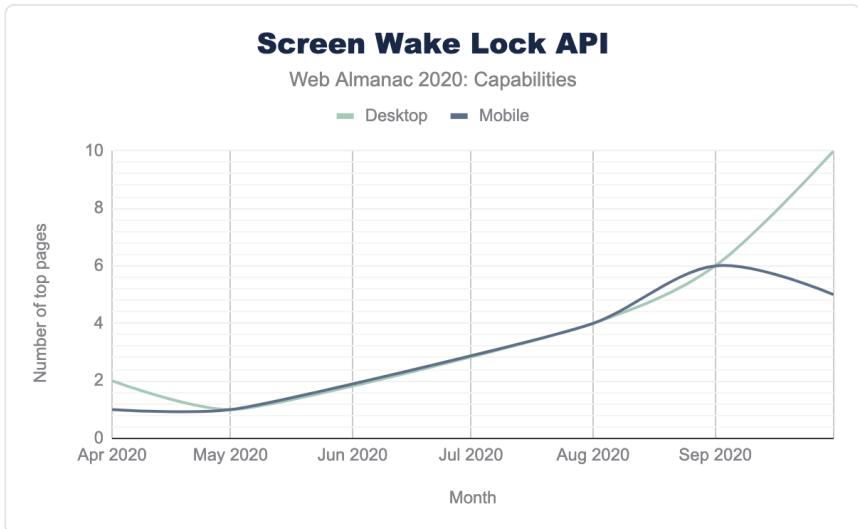


Figure 13.6. Numbers of pages using Screen Wake Lock API.

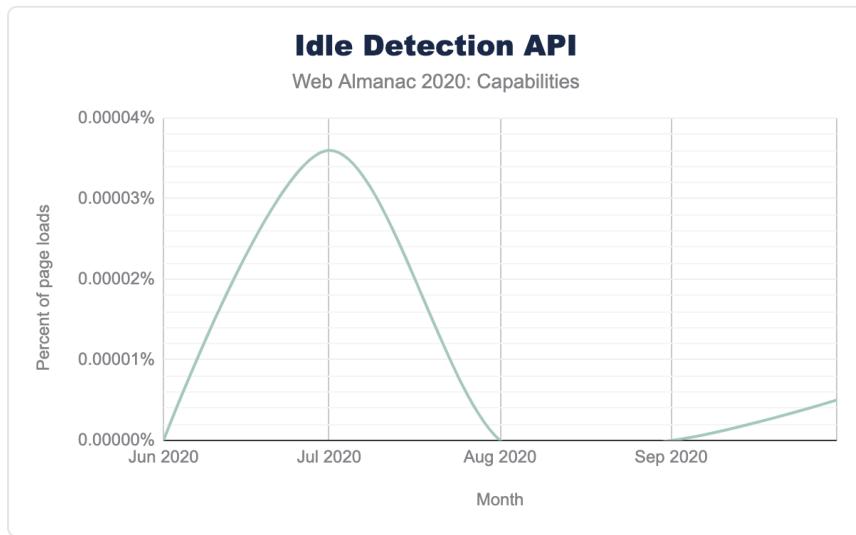
BettyCrocker.com, a popular cooking website in the US, offers their users an option to prevent the screen from going dark while cooking with the help of the Screen Wake Lock API. In a case study, they published that the average session duration was 3.1 times longer than normal, the bounce rate reduced by 50%, and purchase intent indicators increased by about 300%. The interface therefore has a directly measurable effect on the success of the website or application, respectively. The Screen Wake Lock API shipped with Google Chrome 84 in July 2020. The HTTP Archive only has data for April, May, August, September and October. After the release of Chrome 84, usage rose quickly. In October 2020, the API was adopted on 10 desktop and 5 mobile pages.

## Idle Detection API

Some applications need to determine if the user is actively using a device or if they are idle. For instance, chat applications may display that the user is absent. There are various factors that

can be taken into account, such as a lack of interaction with the screen, mouse, or keyboard. The Idle Detection API (WICG Draft Community Group Report) provides an abstract API that allows developers to check if either the user is idle or the screen locked, given a certain threshold.

To do so, the API provides a new `IdleDetector` interface on the global `window` object. Before developers can use this functionality, they have to request permission by calling `IdleDetector.requestPermission()` first. If the user grants the permission, developers can create a new instance of `IdleDetector`. This object provides two properties: `userState` and `screenState`, containing the respective states. It will raise a `change` event when either the user's or the screen's state change. Finally, the idle detector needs to be started by calling its `start()` method. The method takes a configuration object with two parameters: a `threshold` defining the time in milliseconds that the user has to be idle (the minimum is a minute), and developers can optionally pass an `AbortSignal` to the `abort` property, which serves to abort idle detection later on.



*Figure 13.7. Percentage of page loads in Chrome using Idle Detection API.  
(Source: Idle Detection)*

At the time of this writing, the Idle Detection API is in an origin trial, so its API shape may change in the future. For the same reason, its usage is very low and hardly measurable.

## Periodic Background Sync API

When the user closes a web application, it cannot communicate with its backend service

anymore. In some cases, developers might still want to synchronize data on a more or less regular basis, just as native applications can. For instance, news applications might want to download the latest headlines before the user wakes up. The Periodic Background Sync API (WICG Draft Community Group Report) strives to bridge this gap between web and native.

## Register for periodic sync

The Periodic Background Sync API relies on Service Workers that can run even when the app is closed. As with other capabilities, this API requires users' permission first. The API implements a new interface called `PeriodicSyncManager`. If present, developers can access an instance of this interface on the Service Worker's registration. To synchronize data in the background, the application has to register first, by calling `periodicSync.register()` on the registration. This method takes two parameters: a `tag`, which is an arbitrary string to recognize the registration again later on, and a configuration object that takes a `minInterval` property. This property defines the desired minimum interval in milliseconds by the developer. However, the browser ultimately decides how often it will actually invoke background synchronization:

```
registration.periodicSync.register('articles', {
  minInterval: 24 * 60 * 60 * 1000 // one day
});
```

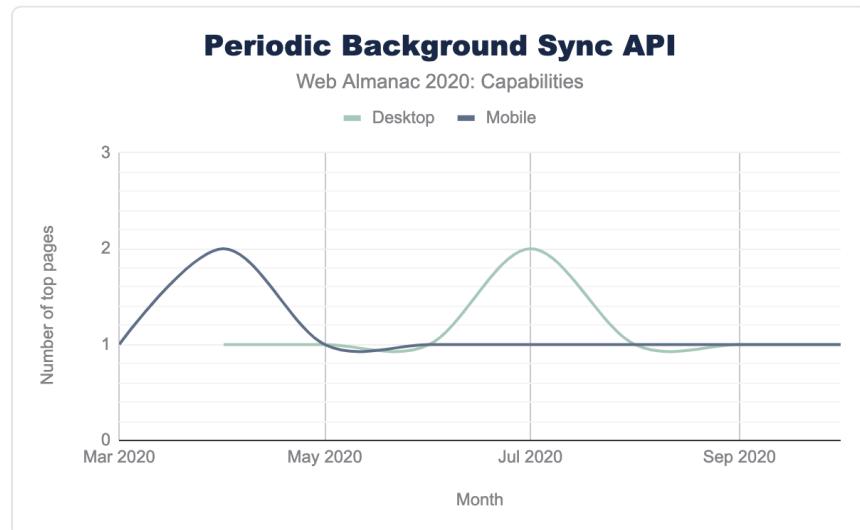
## Respond to a periodic sync interval

For each tick of the interval, and if the device is online, the browser triggers the Service Worker's `periodicsync` event. Then, the Service Worker script can perform the necessary steps to synchronize the data:

```
self.addEventListener('periodicsync', (event) => {
  if (event.tag === 'articles') {
    event.waitUntil(syncStuff());
  }
});
```

At the time of this writing, only Chromium-based browsers implement this API. On these browsers, the application has to be installed first (i.e., added to the homescreen) before the API can be used. The site engagement score of the website defines if and how often periodic sync events can be invoked. In the current conservative implementation, websites can sync content

once a day.



*Figure 13.8. Number of pages using Periodic Background Sync API.*

The use of the interface is currently very low. Over 2020, only one or two pages monitored by HTTP Archive made use of this API.

## Integration with native app stores

PWAs are a versatile application model. However, in some cases, it may still make sense to offer a separate native application: for example, if the app needs to use features that are not available on the web, or based on the programming experience of the app developer team. When the user already has a native app installed, apps might not want to send notifications twice or promote the installation of a corresponding PWA.

To detect if the user already has a related native application or PWA on the system, developers can use the `getInstalledRelatedApps()` method (WICG Draft Community Group Report) on the `navigator` object. This method is currently provided by Chromium-based browsers and works for both Android and Universal Windows Platform (UWP) apps. Developers need to adjust the native app bundles to refer to the website and add information about the native app(s) to the Web App Manifest of the PWA. Calling the `getInstalledRelatedApps()` method will then return the list of apps installed on the user's device:

```
const relatedApps = await navigator.getInstalledRelatedApps();
relatedApps.forEach((app) => {
  console.log(app.id, app.platform, app.url);
});
```

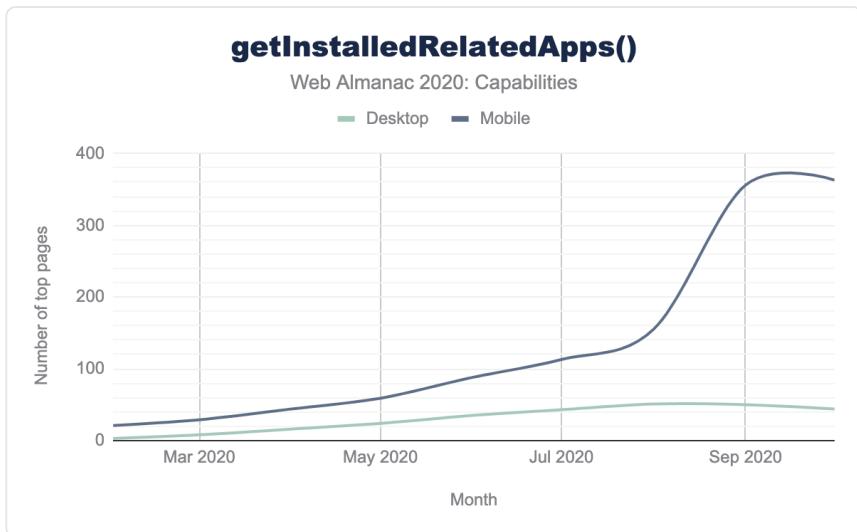


Figure 13.9. Number of pages using `getInstalledRelatedApps()`.

Over the course of 2020, the `getInstalledRelatedApps()` API shows a steady growth on mobile websites. In October, 363 mobile pages tracked by the HTTP Archive made use of this API. On desktop pages, the API does not grow quite as fast. This could also be due to Android stores currently providing significantly more apps than the Microsoft Store does for Windows.

## Content Indexing API

Web apps can store content offline using various ways, such as Cache Storage, or IndexedDB. However, for users it's hard to discover which content is available offline. The Content Indexing API (WICG Editor's Draft) allows developers to expose content more prominently. Currently, Chrome on Android is the only browser to support this API. This browser shows a list of "Articles for you" in the Downloads menu. Content indexed via the Content Indexing API will appear there.

The Content Indexing API extends the Service Worker API by providing a new `ContentIndex` interface. This interface is available via the `index` property of the Service Worker's

registration. The `add()` method allows developers to add content to the index: Each piece of content must have an ID, a URL, a launch URL, title, description, and a set of icons. Optionally, the content can be grouped into different categories such as articles, homepages, or videos. The `delete()` method allows for removing content from the index again, and the `getAll()` method returns a list of all indexed entries.

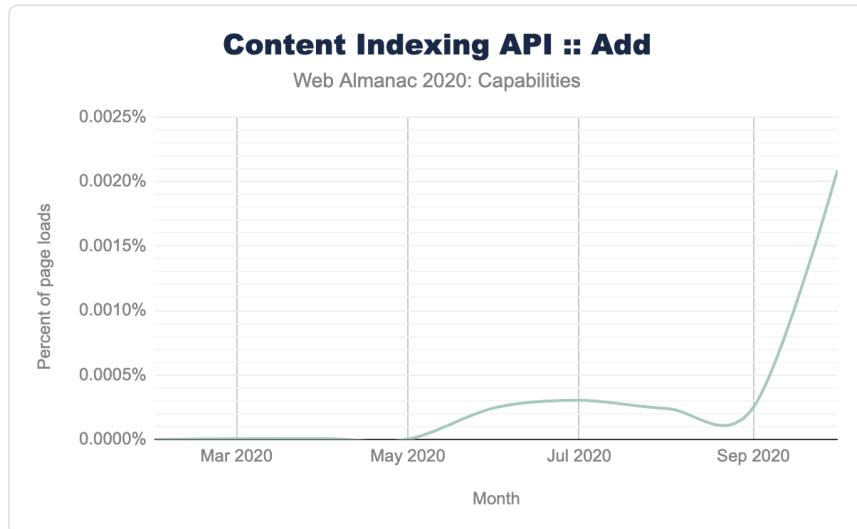


Figure 13.10. Percentage of page loads in Chrome using Content Indexing API.  
(Source: Content Indexing)

The Content Indexing API launched with Chrome 84 in July 2020. Directly after shipping, the API was used during approximately 0.0002% of page loads in Chrome. In October 2020, this value has increased almost tenfold.

## New Transport APIs

Finally, there are two new transport methods that are currently in origin trial. The first one allows developers to receive high-frequency messages with WebSockets, while the second one introduces an entirely new bidirectional communication protocol apart from HTTP and WebSockets.

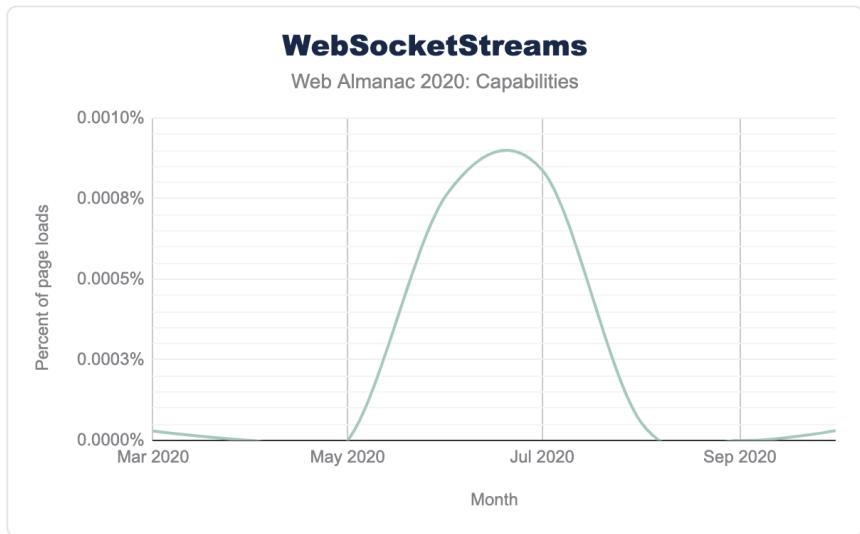
### Backpressure for WebSockets

The WebSocket API is a great choice for bidirectional communication between websites and servers. However, the WebSocket API does not allow for backpressure, so applications dealing

with high-frequency messages may freeze. The `WebSocketStream` API (Explainer, not on the standards track yet) wants to bring easy-to-use backpressure support to the `WebSocket` API by extending it with streams. Instead of using the usual `WebSocket` constructor, developers need to create a new instance of the `WebSocketStream` interface. The `connection` property of the stream returns a promise that resolves to a readable and writable stream that allow to obtain a stream reader or writer, respectively:

```
const wss = new WebSocketStream(WSS_URL);
const {readable, writable} = await wss.connection;
const reader = readable.getReader();
const writer = writable.getWriter();
```

The `WebSocketStream` API transparently solves backpressure, as the stream readers and writers will only proceed if it's safe to do so.



*Figure 13.11. Percentage of page loads in Chrome using `WebSocketStreams`.  
(Source: `WebSocketStream`)*

The `WebSocketStream` API has completed its first origin trial and is now back in the experimentation phase again. This also explains why the usage of this API currently is so low that it's hardly measurable.

## Make it QUIC

QUIC (IETF Internet-Draft) is a multiplexed, stream-based, bidirectional transport protocol implemented on UDP. It's an alternative to HTTP/WebSocket APIs that are implemented on top of TCP. The QuicTransport API is the client-side API for sending messages to and receiving messages from a QUIC server. Developers can choose to send data unreliable via datagrams, or reliably by using its streams API:

```
const transport = new QuicTransport(QUIC_URL);
await transport.ready;

const ws = transport.sendDatagrams();
const stream = await transport.createSendStream();
```

QuicTransport is a valid alternative to WebSockets, as it supports the use cases from the WebSocket API and adds support for scenarios where minimal latency is more important than reliability and message order. This makes it a good choice for games and applications dealing with high-frequency events.

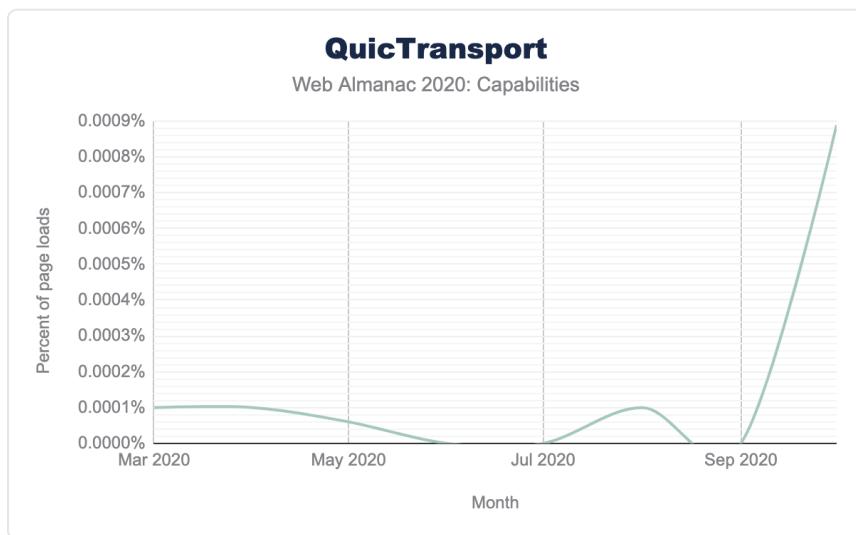


Figure 13.12. Percentage of page loads in Chrome using QuicTransport.  
(Source: QuicTransport)

The use of the interface is currently still so low that it's hardly measurable. In October 2020, it has increased strongly and is now used during 0.00089% of page loads in Chrome.

## Conclusion

The state of web capabilities in 2020 is healthy, as new, powerful APIs regularly ship with new releases of Chromium-based browsers. Some interfaces like the Content Indexing API or Idle Detection API help to add finishing touches to certain web applications. Other APIs, such as the File System Access and Async Clipboard API, allow a whole new application category, namely productivity apps, to finally fully make the shift to the web. Some APIs such as Async Clipboard and Web Share API have already made their way into other, non-Chromium browsers. Safari even was the first mobile browser to implement the Web Share API.

Through its rigorous process, the Fugu team ensures that access to these features takes place in a secure and privacy-friendly manner. Additionally, the Fugu team actively solicits the feedback from other browser vendors and web developers. While the usage of most of these new APIs is comparatively low, some APIs presented in this chapter show an exponential or even hockey stick-like growth, such as the Badging or Content Indexing API. The state of web capabilities in 2021 will depend on the web developers themselves. The author encourages the community to build great web applications, make use of the powerful APIs in a backwards-compatible manner, and help make the web a more capable platform.

### Author



#### Christian Liebel

Twitter: [@christianliebel](https://twitter.com/christianliebel) GitHub: [@christianliebel](https://github.com/christianliebel) Website: <https://christianliebel.com>

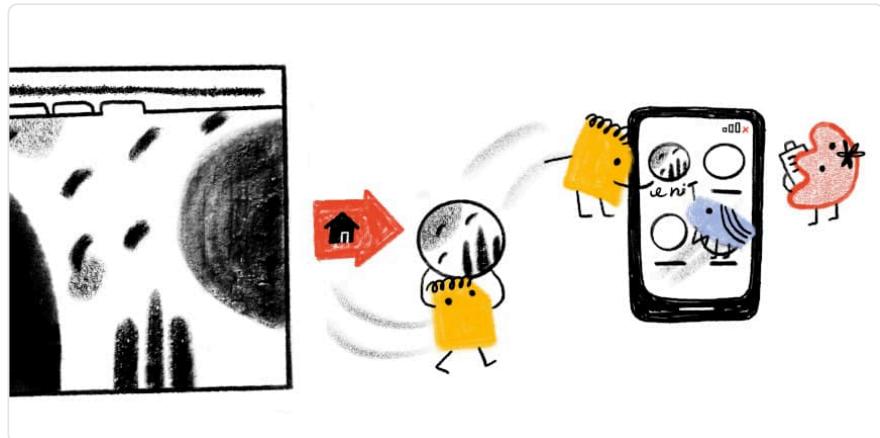
Christian Liebel is a consultant at Thinktecture<sup>27</sup>, supporting clients from various business areas in implementing first-class web applications. He is a Microsoft MVP for Developer Technologies, Google GDE for Web/Capabilities and Angular, and participates in the W3C Web Applications Working Group.

27. <https://thinktecture.com>



## Part II Chapter 14

# PWA [UNEDITED]



Written by [hemanth.hm](#)

Reviewed by [Pascal Schilp](#), [Jad Joubran](#), [Pearl Latteier](#), and [Gokulakrishnan Kalaikovan](#)

Analyzed by [Barry Pollard](#)

## Introduction

In 1990 we had the first ever browser called the “WorldWideWeb” and ever since the web and the browser have been evolving and for the web to progress itself into a native behaviour is a big win especially in this era of mobile domination. URLs and web browsers have provided a ubiquitous way to distribute information and so a technology which provides native app capabilities to the browser is a game changer. Progressive Web Apps provide such advantages for the web to compete with other applications.

Simply put, a web application which give native-like application experience can be considered as a PWA,. It is built using common web technologies including HTML, CSS and JavaScript and can operate seamlessly across devices and environments on a standards-compliant browser.

The crux of a progressive web app is the *service worker*, which can be thought of as a proxy sitting between the browser and user. A service worker gives the developer total control over the network, rather than the network controlling the application.

As last year's chapter stated, it started in December 2014 when Chrome 40 first implemented the meat of what is now known as Progressive Web Apps (PWA). This was a collaborative effort for the web standards body and the term PWA was coined by Frances Berriman and Alex Russell in 2015.

In this chapter of Web Almanac we will be looking into each of the components that make PWA what it is, from a data-driven perspective.

## Service workers

Service workers are at the very center of the progressive web apps and can be thought as a proxy servers between web applications running in browsers and the network. They help developers control the network requests rather than the network requests controlling the application.

### Service worker usage

From the data we gathered it was derived that about **0.88%** desktop sites and **0.87%** mobile sites use a service worker. This was for the month of August 2020 and, to put that into perspective, that equates to 49,305 (out of 5,593,642) desktop sites and 55,019 (out of 6,347,919) mobile sites. While that usage may seem low, it is important that we realize that other measurements equate that to **16.6%** of the web traffic—the difference being due to high traffic websites tending to use service workers more.

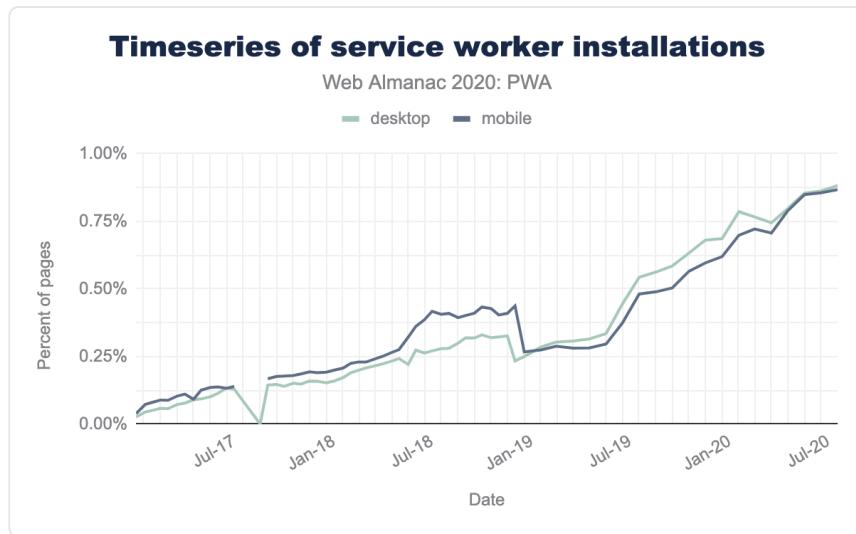


Figure 14.1. Timeseries of Service Worker installation.

## Lighthouse insights

Lighthouse provides automated auditing, performance metrics, and best practices for the web and has been instrumental in shaping web's performance. We looked at the PWA category audits gathered for over 6,811,475 pages and this has given us great insights on a few important touch points.

<b>Lighthouse Audit</b>	<b>Weight</b>	<b>Percentage</b>
<i>load-fast-enough-for-pwa</i>	7	27.97%*
<i>works-offline</i>	5	0.86%
<i>installable-manifest</i>	2	2.21%
<i>is-on-https</i>	2	66.67%
<i>redirects-http</i>	2	70.33%
<i>viewport</i>	2	88.43%
<i>apple-touch-icon</i>	1	34.75%
<i>content-width</i>	1	79.37%
<i>maskable-icon</i>	1	0.11%
<i>offline-start-url</i>	1	0.75%
<i>service-worker</i>	1	1.03%
<i>splash-screen</i>	1	1.90%
<i>themed-omnibox</i>	1	4.00%
<i>without-javascript</i>	1	97.57%

Figure 14.2. Lighthouse PWA audits.

Note the Lighthouse performance stats were incorrect for our August crawl so the *load-fast-enough-for-pwa* stat has been replaced with September results.

A fast page load ensures a good mobile user experience, particularly when slower cellular network's are taken into consideration.

**27.56%** of pages loaded fast enough for a PWA. Given how geographically distributed the web is, having a fast load time with lighter pages matter the most of the next billion users of the web, most of whom will be introduced to the internet via a mobile device.

If you're building a Progressive Web App, consider using a service worker so that your app can work offline **0.92%** of pages were offline ready.

Browsers can proactively prompt users to add your app to their homescreen, which can lead to higher engagement. **2.21%** of pages had an Installable manifest. Manifest plays an important role in how the application starts, the looks and feel of the icon on the homescreen and as an impact on the engagement rate directly.

All sites should be protected with HTTPS, even ones that don't handle sensitive data. This includes avoiding mixed content, where some resources are loaded over HTTP despite the initial request being served over HTTPS. HTTPS prevents intruders from tampering with or passively listening in on the communications between your app and your users, and is a prerequisite for HTTP/2 and many new web platform APIs. Learn more about is-on-https check. **67.27%** of sites were on HTTPS and it is surprising that we haven't reached there yet. This number is pretty decent and will get better as browsers mandate the applications to be on HTTPS and scrutinize those which are not on HTTPS.

If you've already set up HTTPS, make sure that you redirect all HTTP traffic to HTTPS in order to enable secure connection the users without changing the URL **69.92%** of the sites redirects HTTP. Redirecting all the HTTP to HTTPS on your application should be simple steps towards robustness, though the HTTP redirection to HTTPS has a decent number, it can do better.

By adding `<meta name="viewport">` tag to optimize your app for mobile screens. **88.43%** of the sites have the viewport meta tag. It is not surprising that the usage of viewport meta tag is on the higher side as most of the applications are aware and getting there in terms of viewport optimization.

For ideal appearance on iOS, your progressive web app should define an `apple-touch-icon` meta tag. It must point to a non-transparent 192px (or 180px) square PNG. **32.34%** of the sites use the apple touch icon.

If the width of your app's content doesn't match the width of the viewport, your app might not be optimized for mobile screens. **79.18%** of the sites have the content-width set.

A maskable icon ensures that the image fills the entire shape without being letterboxed when adding the progressive web app to the home screen. Only **0.11%** of sites use this, but given that it is a brand new feature, having any usage here is encouraging. As it is a new feature we were expecting the numbers to be very low and are expected to improve in the coming years.

A service worker enables your web app to be reliable in unpredictable network conditions. **0.77%** of sites has an offline start URL.

The service worker is the feature that enables your app to use many Progressive Web App features, such as offline usage and push notifications. **1.05%** of pages have service workers enabled. Service worker helps to achieve offline support, which is the most important feature for a PWA, as flaky networks are the most common issue that the users of web applications face. Given that this can be addressed with service workers, it is surprising that number is still so low.

A themed splash screen ensures a native like experience when users launch your app from their homescreens. **1.95%** of pages had splash screens.

The browser address bar can be themed to match your site. **3.98%** of pages had themed omnibox.

Your app should display some content when JavaScript is disabled, even if it's just a warning to the user that JavaScript is required to use the app. **96.23%** pages can work with JavaScript disabled.

## Service worker events

In a service worker one can listen for a number of events:

1. `install`, which occurs upon service worker installation.
2. `activate`, which occurs upon service worker activation.
3. `fetch`, which occurs whenever a resource is fetched.
4. `push`, which occurs when a push notification arrives.
5. `notificationclick`, which occurs when a notification is being clicked.
6. `notificationclose`, which occurs when a notification is being closed.
7. `message`, which occurs when a message sent via `postMessage()` arrives.
8. `sync`, which occurs when a background sync event occurs.

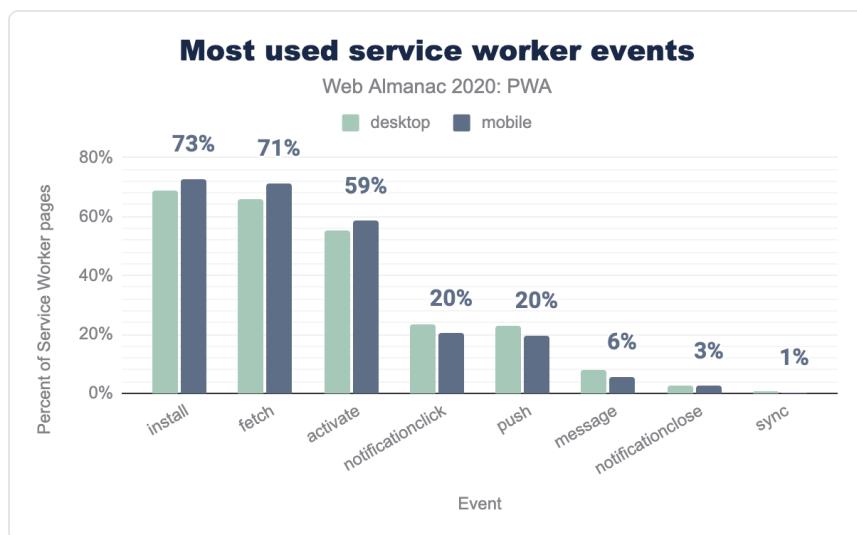


Figure 14.3. Most used service worker events.

We have examined which of these events are being listened to by service workers in our data set. The results for mobile and desktop are very similar with `install`, `fetch`, and `activate` being the three most popular events, followed by `message`, `notification click`, `push` and `sync`. If we interpret these results, offline use cases that service workers enable are the most attractive feature for app developers, far ahead of push notifications. Due to its limited

availability, and less common use case, background sync doesn't play a significant role at this time.

## Web app manifests

The web app manifest is a JSON-based file that provides developers with a centralized place to put metadata associated with a web application, it dictates how the application should behave on desktop or mobile in terms of the icon, orientation, theme color and likes.

Having a web app manifest does not necessarily indicate the site is a progressive web app, as they can exist independently of service worker usage. However, as we are interesting PWAs in this chapter, we have investigated only those manifests for sites where a service worker also exists. This is different than the approach taken in last year's PWA chapter which looked at overall manifest usage so you may notice some differences in results this year.

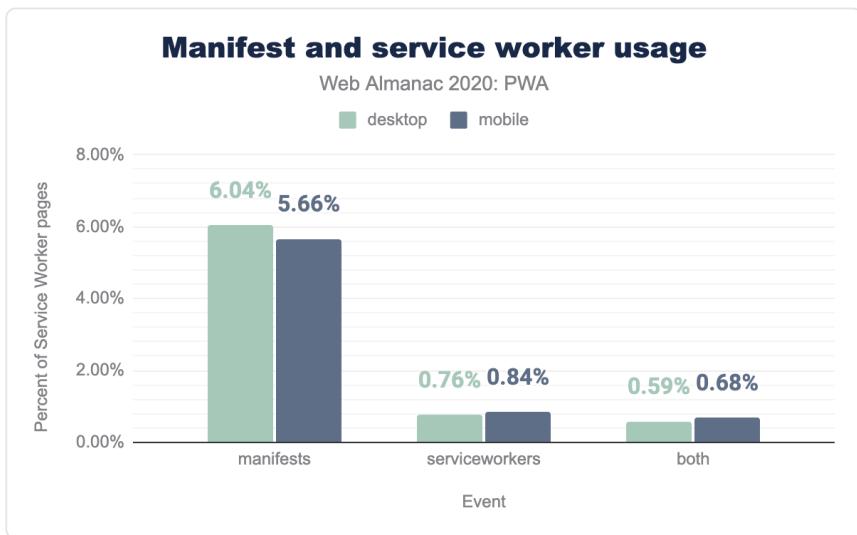


Figure 14.4. Manifest and service worker usage.

## Manifest Properties

Web manifest dictates the applications meta properties. We looked at the different properties defined by the Web App Manifest specification, and also considered non-standard proprietary properties. According to the spec, the following properties are valid properties:

1. `background_color`

- 
- 2. categories
  - 3. description
  - 4. dir
  - 5. display
  - 6. iarc\_rating\_id
  - 7. icons
  - 8. lang
  - 9. name
  - 10. orientation
  - 11. prefer\_related\_applications
  - 12. related\_applications
  - 13. scope
  - 14. screenshots
  - 15. short\_name
  - 16. shortcuts
  - 17. start\_url
  - 18. theme\_color

There were very little differences between mobile and desktop stats.

The proprietary properties we encountered frequently were `gcm_sender_id` used by Google Cloud Messaging (GCM) service. We also found other interesting attributes like: `browser_action`, `DO_NOT_CHANGE_GCM_SENDER_ID` (which was basically a comment, used as JSON doesn't allow comments), `scope`, `public_path`, `cacheDigest`.

On both platforms, however, there's a long tail of properties that are not interpreted by browsers yet contain potentially useful metadata.

We also found a non-trivial amount of mistyped properties; our favorite ones being variation of `theme-color`, `Theme_color`, `theme-color`, `Theme_color` and `oriendation`.

In order for a PWA to be fruitful it needs to have a manifest and a service worker. It is interesting to note that manifests are used a lot more than service workers. This is due, in large part, to the fact that CMS like WordPress, Drupal and Joomla have manifests by default.

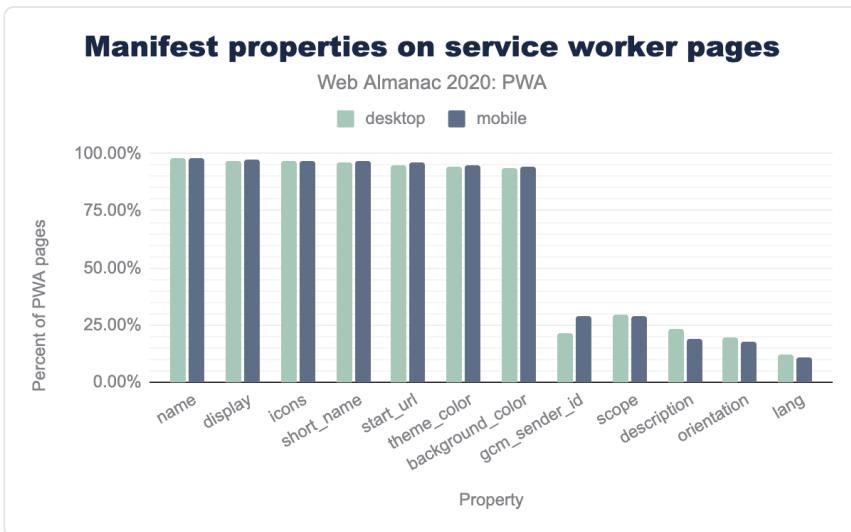


Figure 14.5. Manifest properties on service worker pages.

## Top Manifest display values

Out of the five most common `display` values, `standalone` dominated the list with 86.73% of desktop and 89.28% of mobile pages using this. This isn't surprising at all as this mode provides the native app-like feel. Next in the list was `minimal-ui` with 6.30% of desktop and 5.00% of mobile sites opting for them. This is similar to `standalone` except for the fact that some browser UI is retained.

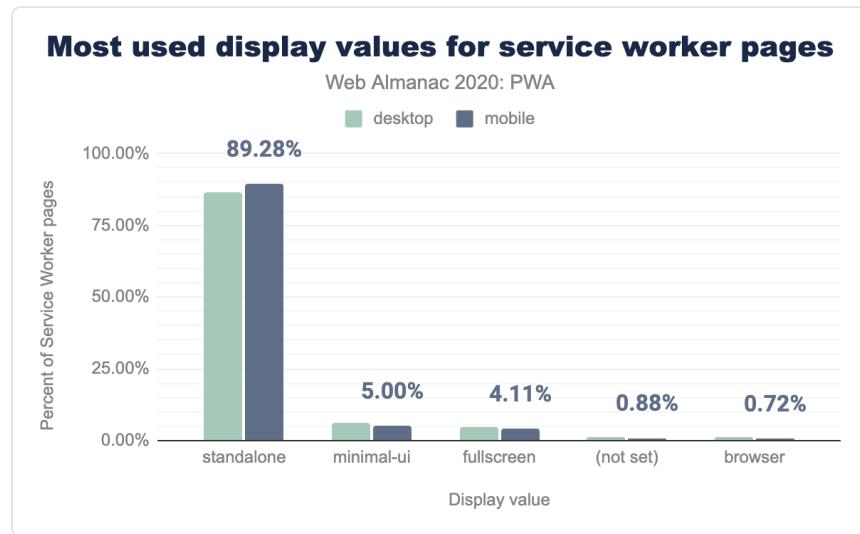


Figure 14.6. Most used `display` values for service worker pages.

## Top manifest categories

Out of all the top categories, shopping stood at the top at with 13.16% on the mobile traffic, which is not unexpected as PWAs are e-commerce applications. News was next with 5.26% on the mobile traffic.

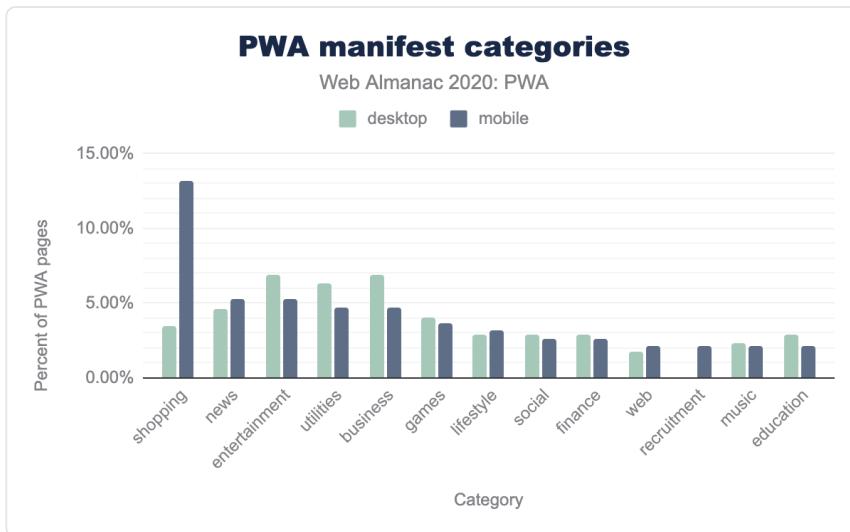


Figure 14.7. PWA manifest categories.

## Manifests preferring native

98.24% and 98.52% of mobile sites set the `preferred_related_applications` manifest property to not prefer native apps, but instead use web version where they exist. For the small percentage where this is set to `true` this is a signal that there are many web applications that just have a manifest but aren't really full PWAs yet.

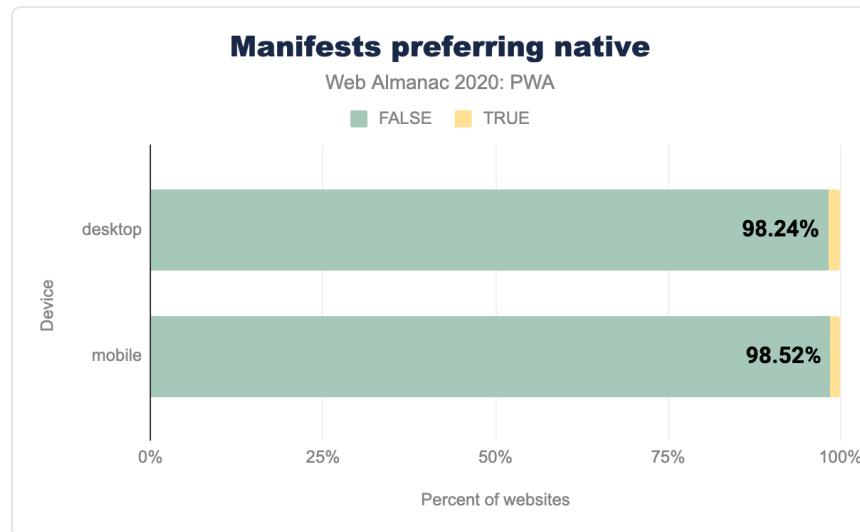


Figure 14.8. Manifest preferring native.

## Top manifest icon sizes

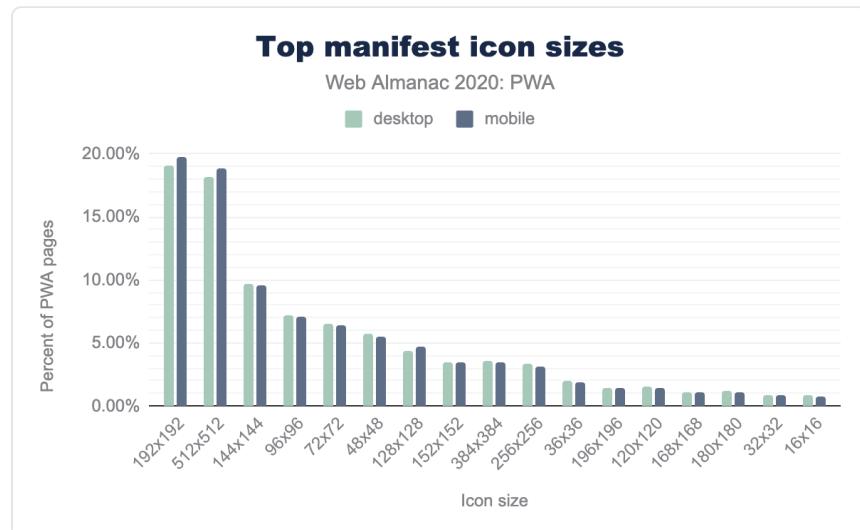


Figure 14.9. Top manifest icon sizes.

Lighthouse requires at least an icon sized 192x192 pixels, but common favicon generation tools

create a plethora of other sizes, too. It is always better to use the recommended icon sizes for each device so it is encouraging to see such a wide spread usage of different icon sizes.

## Top manifest orientations

The valid values for the orientation property are defined in the Screen Orientation API specification. Currently, they are:

1. any
2. natural
3. landscape
4. portrait
5. portrait-primary
6. portrait-secondary
7. landscape-primary
8. landscape-secondary

Out of which we noticed that `portrait`, `any` and `portrait-primary` properties took precedence.

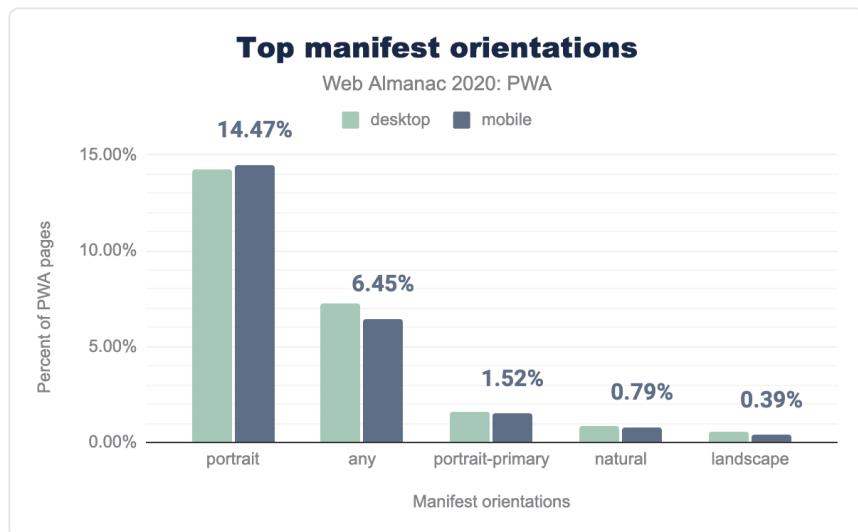


Figure 14.10. Top manifest orientations.

## Service worker libraries

There are many cases, where the service workers use libraries as dependencies, be it external dependencies or the application's internal dependencies. These are usually fetched to the service worker via `importScripts` API, in this section we will look into stats on such libraries.

### Popular import scripts

The `importScripts()` API of the `WorkerGlobalScope` interface synchronously imports one or more scripts into the worker's scope, the same is used to import external dependencies to the service worker.

<b>client</b>	<b>desktop</b>	<b>mobile</b>
<code>Uses Importscript</code>	29.60%	23.76%
<code>Workbox</code>	17.70%	15.25%
<code>sw_toolbox</code>	13.92%	12.84%
<code>firebase</code>	3.40%	3.09%
<code>OneSignalSDK</code>	4.23%	2.76%
<code>najava</code>	1.89%	1.52%
<code>upush</code>	1.45%	1.23%
<code>cache_polyfill</code>	0.70%	0.72%
<code>analytics_helper</code>	0.34%	0.39%
<code>Other Library</code>	0.27%	0.15%
<code>No Library</code>	58.81%	64.44%

Figure 14.11. PWA library usage.

Around **30%** of the desktop and **25%** of mobile sites uses `importScripts`, of which `workbox`, `sw_toolbox` and `firebase` take the first three positions respectively.

### Workbox usage

Out of many libraries available, Workbox was the most heavily used with an adoption rate of

12.86% and 15.29% of PWA sites on mobile and desktop respectively.

Out of many methods that Workbox provides, we noticed that `strategies` were used by 29.53% of desktop and 25.71% on mobile, `routing` followed it with 18.91% and 15.61% adoption and finally `precaching` were next most used with 16.54% and 12.98% on desktop and mobile respectively.

This indicated that the strategies API, as one of the most complicated requirements for the developers, played a very important role when they decided to code themselves or rely on libraries like Workbox.

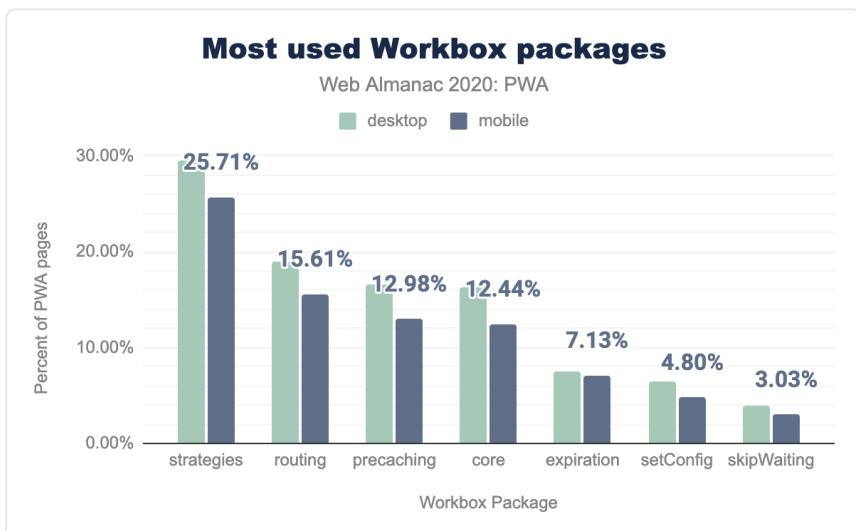


Figure 14.12. Most used Workbox packages.

## Conclusion

The stats in this chapter show that PWAs are still continuing to grow in adoption, due to the advantages they give for performance and greater control over caching particularly for mobile. With those advantages and ever increasing capabilities, means we still have a lot of potential for growth and when compared to 2019. We expect to see even more progress in 2021!

More and more browsers and platforms are supporting the technologies powering PWAs. This year, we saw that Edge gained support for the Web App Manifest. Depending on your use case and target market, you may find that the majority of your users (close to 96%) have PWA support. That is a great improvement! In all cases, it's important to approach technologies such as Service Worker, Web App Manifest should be treated as progressive enhancement. Where

you can provide an exceptional user experience through these technologies whenever possible. With the above stats, we're excited for another year of PWA growth!

## Author

---



**hemanth.hm**

 @gnumanth  hemanth  <http://h3manth.com>

Hemanth HN<sup>28</sup> is a FOSS Computer polyglot, FOSS philospher, GDE for web and payments domain, DuckDuckGo community member, TC39 delegate and Google Launchpad Accelerator mentor. Loves The WEB && CLI. Hosts TC39er.us<sup>29</sup> podcast.

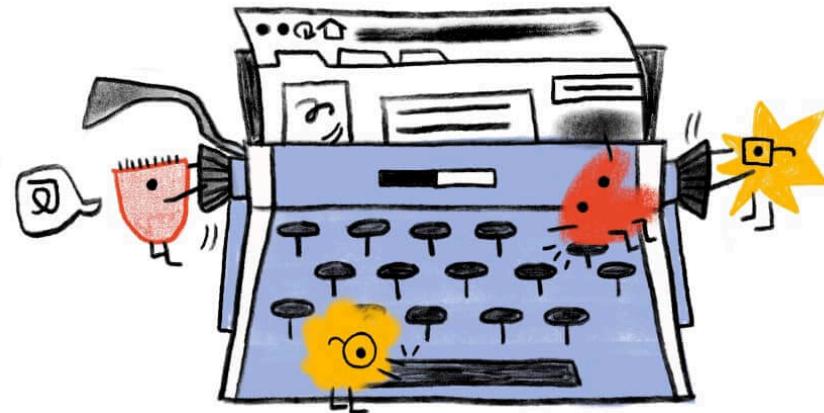
---

28. <https://h3manth.com>

29. <https://TC39er.us>

# Part III Chapter 15

# CMS



*Written by Alex Denning*

*Reviewed by Jonathan Wold, Renee Johnson, and Alberto Medina*

*Analyzed by Greg Brimble and Rick Viscomi*

## Introduction

The term Content Management System (CMS) refers to systems enabling individuals and organizations to create, manage, and publish content. A CMS for web content, specifically, is a system aimed at creating, managing, and publishing content to be consumed and experienced via the internet.

Each CMS implements some subset of a wide range of content management capabilities and the corresponding mechanisms for users to build websites easily and effectively around their content. Content is often stored in a type of database, providing users with the flexibility to reuse it wherever needed for their content strategy. CMSs also provide administrative capabilities aimed at making it easy for users to upload and manage content as needed.

There is great variability on the type and scope of the support CMSs provide for building sites; some provide ready-to-use templates which are supplemented with user content, and others require much more user involvement for designing and constructing the site structure.

When we think about CMSs, we need to account for all the components that play a role in the viability of such a system for providing a platform for publishing content on the web. All of these components form an ecosystem surrounding the CMS platform, and they include hosting providers, extension developers, development agencies, site builders, etc. Thus, when we talk about a CMS, we usually refer to both the platform itself and its surrounding ecosystem.

There are many interesting and important aspects to analyze and questions to answer in our quest to understand the CMS space and its role in the present and the future of the web. We acknowledge the vastness and complexity of the CMS platforms space and bring to it our curiosity along with deep expertise on some of the major players in the space.

In this chapter, we seek to help understand the current state of the CMS ecosystems, the role they play in shaping users' perception of how content can be consumed and experienced on the web, and their impact on the environment. Our goal is to discuss aspects related to the CMS landscape in general, and the characteristics of web pages generated by these systems.

This second edition of the Web Almanac builds on last year's work. We now have the benefit of being able to compare the 2020 results to 2019 in order to start establishing trends. Let's dive into our analysis.

## Why use a CMS in 2020?

People and organizations use a CMS in 2020 as in many cases CMSs offer a shortcut to creating a website which meets their needs. As we'll discuss later, there are both general and specialized CMSs. The general CMSs are often extensible through add-ons, and the specialized CMSs are often focused on specific industry needs or functionality.

Whichever CMS used, it is in use because it solves a problem for the user or organization. It's beyond our scope to explore why each CMS is chosen, but later we do explore why the most popular CMS, WordPress, is disproportionately chosen.

## CMS adoption

Our analysis throughout this work looks at desktop and mobile websites. The vast majority of URLs we looked at are in both datasets, but some URLs are only accessed by desktop or mobile devices. This can cause small divergences in the data, and we thus look at desktop and mobile results separately.

More than 42% of web pages are powered by a CMS platform, an increase of over 5% from 2019. This breaks down to 42.18% on desktop, up from 40.01% in 2019, and 42.27% on mobile, up from 39.61% in 2019.

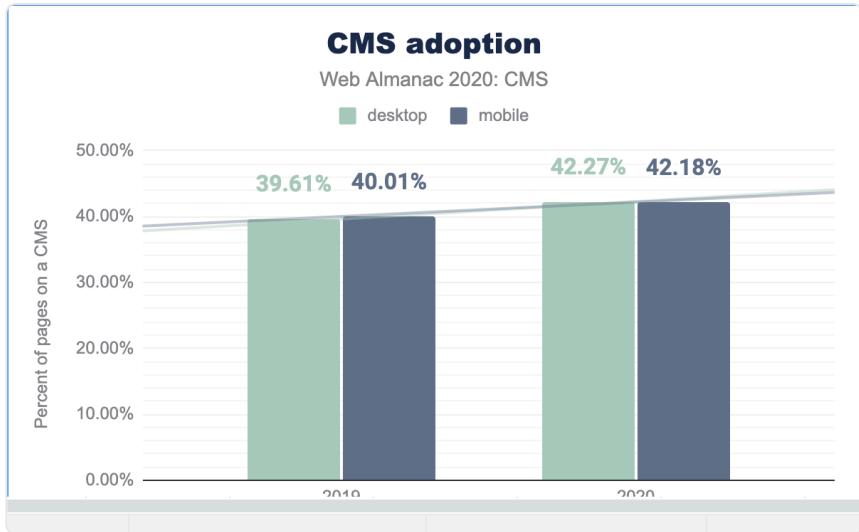


Figure 15.1. CMS adoption trend.

Year	Desktop	Mobile
2019	39.61%	40.01%
2020	42.27%	42.18%
% Change	6.71%	5.43%

Figure 15.2. CMS adoption statistics.

The increase in desktop web pages powered by a CMS platform is 5.43% from last year. On mobile this increase is roughly a quarter higher, at 6.71%.

As with last year, we see different results from other datasets for tracking market share of CMS platforms, such as W3Techs. W3Techs reports at the time of writing that 60.6% of web pages are created by CMSs, up from 56.4% a year ago. This is a 6.4% increase, which broadly matches our findings.

The deviation between our analysis and W3Techs' analysis can be explained by a difference in research methodologies. You can read more about ours on the [Methodology page](#).

Our research identified 222 individual CMSs, with these ranging from a single install to millions on a single CMS.

Some of them are open source (e.g. WordPress, Joomla, others) and some of them are proprietary (e.g. Wix, Squarespace, others). As we'll discuss later, the top 3 CMSs by adoption

share are all open source, but proprietary platforms have seen large increases in adoption share this year. Some CMS platforms can be used on "free" hosted or self-hosted plans, and there are also options for using these platforms on higher-tiered plans even at the enterprise level.

The CMS space as a whole is a complex, federated universe of CMS ecosystems, all separated and at the same time intertwined. Our research shows CMSs are only getting more important. The minimum of 5% increase in adoption of CMSs shows that in a year when COVID-19 has created immense uncertainty, solid CMS platforms have provided some stability. As we discussed last year, these platforms play a key role for us to succeed in our collective quest for an evergreen, healthy, and vibrant web. This has become truer since, and we expect it to continue to be the case going forward.

## Top CMSs

Our analysis counted 222 separate CMSs. While this is a high count, 204 (92%) of these have an adoption share of 0.01% or lower. This leaves only 13 CMSs with an adoption share of between 0.1 and 1%, and four with a share of between 1 and 2%, and one with a share over this.

The one CMS with a share over 2% is WordPress, which has a 31% usage share. This is over 15 times the share of the next most popular CMS, Joomla:

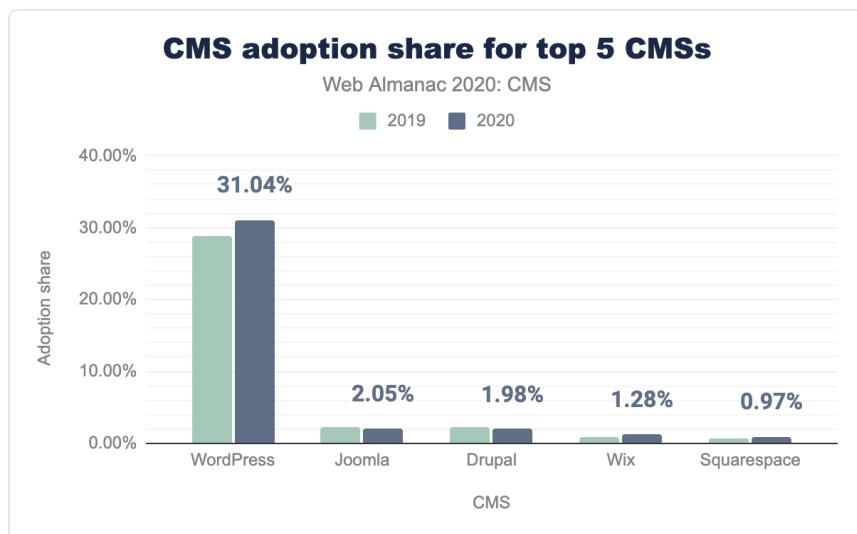
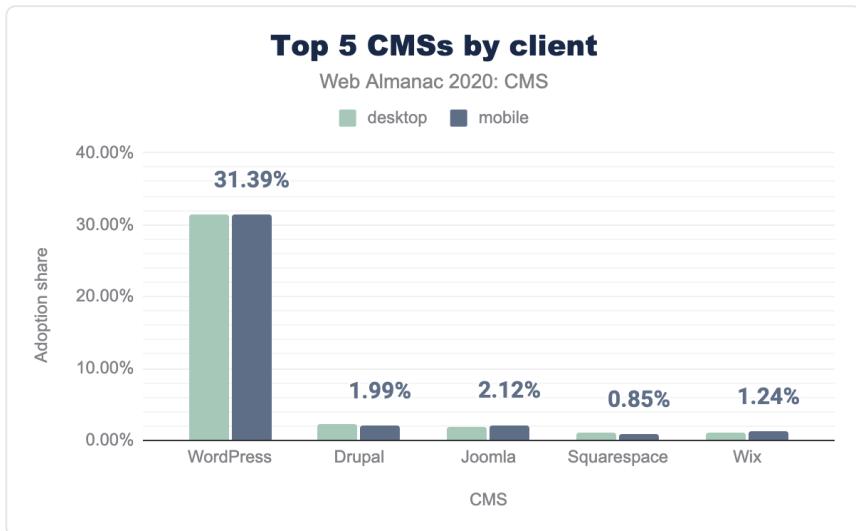


Figure 15.3. CMS adoption share for top 5 CMSs.

Joomla and Drupal have lost 8% and 10% of their adoption share respectively, whilst Wix and Squarespace have gained an extra 41% and 28% adoption share respectively. WordPress has

gained an extra 7% adoption share in the last year, which is a larger absolute increase than the total share for Joomla, the next most popular CMS.

These numbers are broadly consistent when split across desktop and mobile:



*Figure 15.4. Top 5 CMSs by client.*

For WordPress the numbers are very similar; for the other CMSs the difference is larger. Drupal and Squarespace have 16.7 and 26.3% more websites on desktop than mobile respectively, whilst Joomla and Wix have 7.5 and 15.2% more times on mobile than desktop.

The 0.1 to 1% adoption share category sees significantly more movement. These account for CMSs powering up to 50,000 websites.

CMS	2019	2020	% change
WordPress	28.91%	31.04%	7%
Joomla	2.24%	2.05%	-8%
Drupal	2.21%	1.98%	-10%
Wix	0.91%	1.28%	41%
Squarespace	0.76%	0.97%	28%
1C-Bitrix	0.55%	0.61%	10%
TYPO3 CMS	0.53%	0.52%	-2%
Weebly	0.39%	0.33%	-15%
Jimdo	0.28%	0.24%	-16%
Adobe Experience Manager	0.27%	0.23%	-14%
Duda		0.22%	
GoDaddy Website Builder		0.18%	
DNN	0.20%	0.16%	-19%
DataLife Engine	0.19%	0.16%	-12%
Tilda	0.08%	0.16%	100%
Liferay	0.12%	0.11%	-10%
Microsoft SharePoint	0.15%	0.11%	-25%
Kentico CMS	0.00%	0.11%	10819%
Contao	0.09%	0.09%	0%
Craft CMS	0.08%	0.09%	5%
MyWebsite		0.09%	
Concrete5	0.10%	0.09%	-12%

Figure 15.5. Relative % adoption of smaller CMSs (0.1% - 1% adoption share)

We see three new entrants here: Duda, GoDaddy Website Builder, and MyWebsite. Two, Tilda and Kentico CMS, have seen an adoption share change of over 100% in the last year. This "long tail" of CMSs cover a mix of open source and proprietary platforms and include everything from consumer-friendly to industry-specific. An incredible strength of the CMS platforms as a whole

is one can get specialized software which powers every conceivable type of website.

When we look at CMS adoption share relative to other CMSs (thus excluding websites with no CMS), The dominance of WordPress becomes clear. The adoption share of websites with a CMS is 74.2%. With these numbers relative, Joomla, Drupal, Wix, and Squarespace receive higher adoption rates: 4.9%, 4.7%, 3.1%, and 2.3% respectively:

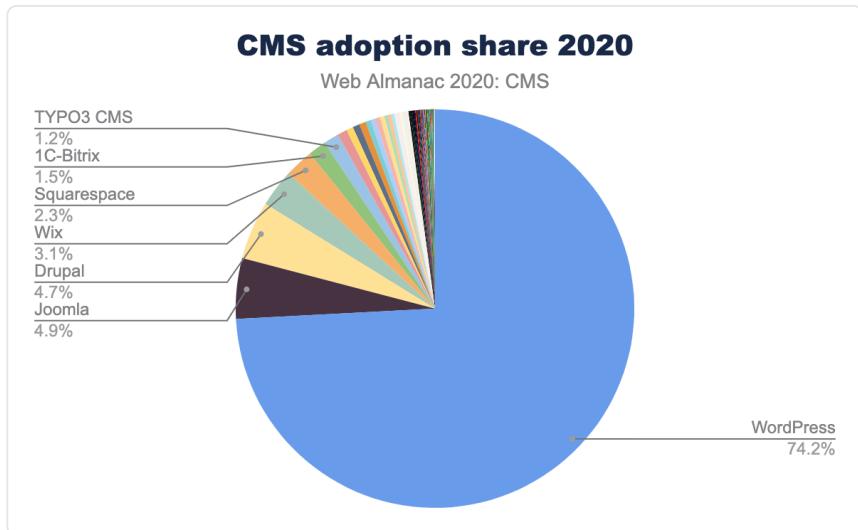


Figure 15.6. CMS adoption share 2020.

## WordPress usage

WordPress dominates this space and thus deserves further discussion.

WordPress is an open source project with a mission to "democratize publishing". The CMS is free. While this is likely an important factor in its adoption share, the two next most popular CMSs—Joomla and Drupal—are also free. The WordPress community, contributors, and business ecosystem are likely the major differentiators.

A "core" WordPress community maintains the CMS and services requirements for additional functionality through custom services and products (themes and plugins). This community has an outsized impact, with a relatively small number of people maintaining both the CMS itself and providing the additional functionality which makes WordPress sufficiently powerful and flexible that it can service most types of website. This flexibility is important when explaining the market share.

Deriving from this flexibility, WordPress also has a low barrier of entry for developers and site

"builders" or "implementers". We see a virtuous cycle: flexible extensions offer ever-easier site building, which lets more and more users build ever-more-powerful sites with WordPress. This increase in users makes it more attractive for developers to create better and better extensions, furthering the cycle.

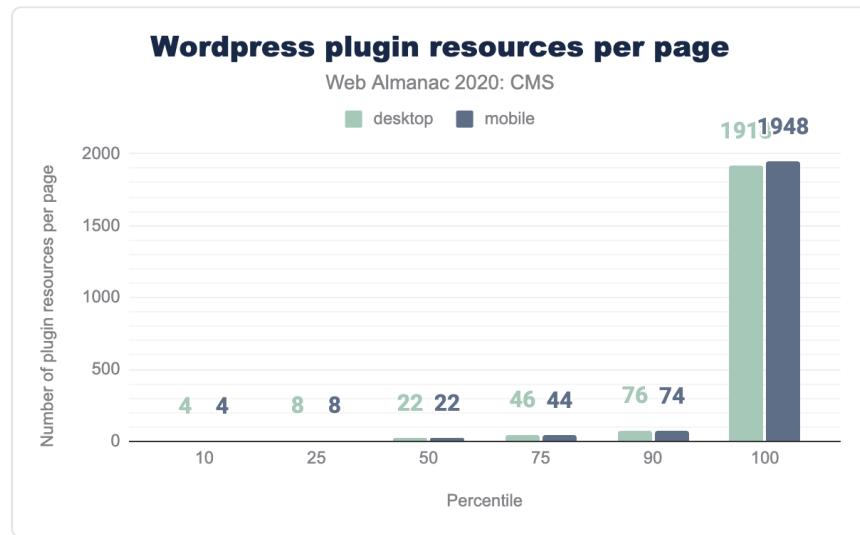


Figure 15.7. WordPress plugin resources per page.

We explored how WordPress sites use these extensions, which are typically WordPress plugins. The median WordPress site (on desktop and mobile) loads 22 plugin resources per page, with sites at the 90th percentile loading 76 and 74 resources per page on desktop and mobile respectively. At the 100th percentile this goes as high as 1918 and 1948 resources per page on desktop and mobile respectively. Whilst we can't compare this to other CMSs, it seems likely that WordPress's extension ecosystem is a major contributor to its high adoption rate.

WordPress's adoption share growth of 7.40% from 2019 to 2020 outstrips the overall increase in adoption of CMSs as a whole. This suggests WordPress has appeal significantly beyond the "average" CMS.

2020 has seen the impact of COVID-19. This may explain the increase in market share. Anecdotally, we can suggest that with many physical businesses closing permanently or temporarily, there has been increased demand for websites in general and WordPress as the largest CMS has benefited from this. Further research in the coming years will be required to ascertain the full impact.

With the adoption share of CMSs explored, let's now turn our attention to user experience.

## CMS user experience

CMSs must offer a good user experience. With so much of the web relying on CMSs to serve pages, it is the responsibility of the CMS at the platform-level to ensure the user experience is good. Our aim is to shed light on real-world user experience when using CMS-powered websites.

To achieve this, we turn our analysis towards some user-perceived performance metrics, which are captured in the three Core Web Vitals metrics, as well as the Lighthouse scores in the SEO and Accessibility categories.

### Chrome User Experience Report

In this section we take a look at three important factors provided by the Chrome User Experience Report, which can shed light on our understanding of how users are experiencing CMS-powered web pages in the wild:

- First Contentful Paint (FCP)
- First Input Delay (FID)
- Cumulative Layout Shift (CLS)

These metrics aim to cover the core elements which are indicative of a great web user experience. The [\[./performance\]](#) chapter will cover these in more detail, but here we are interested in looking at these metrics specifically in terms of CMSs. Let's review each of these in turn.

#### Largest Contentful Paint

Largest Contentful Paint (LCP) measures the point when the page's main content has likely loaded and thus the page is useful to the user. It does this by measuring the render time of the largest image or text block visible within the viewport.

This is different to First Contentful Pain (FCP), which measures from page load until content such as text or an image is first displayed. LCP is regarded as a good proxy for measuring when the main content of a page is loaded.

A "good" LCP is regarded as under 2.5 seconds. The average website on one of the top five CMSs does not have a good LCP. Only Drupal on desktop scores over 50% here. We see major discrepancies between desktop and mobile scores: WordPress is fairly even at 33% on desktop and 25% on mobile, but Squarespace scores 37% on desktop and only 12% on mobile.

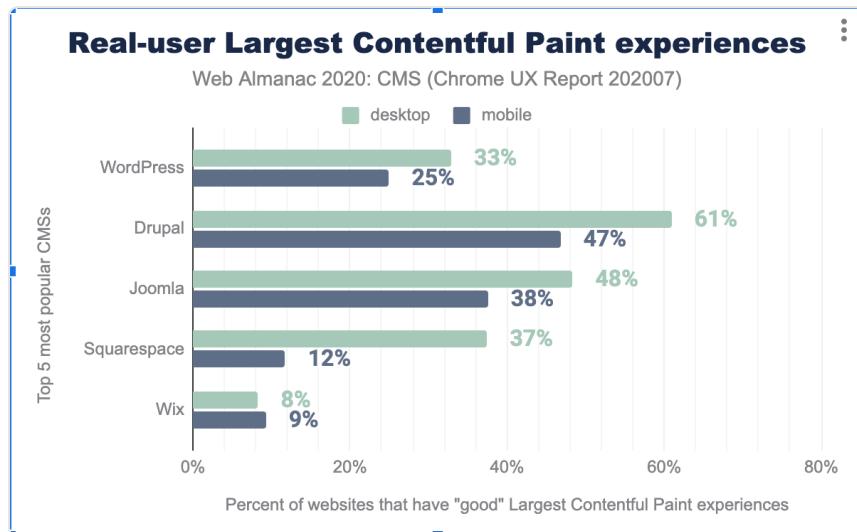


Figure 15.8. Real-user Largest Contentful Paint experiences.

Even though we'd love to see CMSs performing much better here, there are still some positive takeaways from these results. For one, the fact that 61% of Drupal websites have good LCP is especially notable because it's much better than the global distribution of 48% of websites having good LCP, according to the Chrome UX Report. For 1 in 3 or 4 WordPress websites to have good LCP is also kind of amazing, given the sheer magnitude of the number of WordPress websites. Wix does have some catching up to do, but it's encouraging to see that Wix engineers are actively working on fixing performance issues, so this will be something to keep an eye on over the years.

## First Input Delay

First Input Delay (FID) measures the time from when a user first interacts with your site (i.e. when they click a link, tap on a button, or use a custom, JavaScript-powered control) to the time when the browser is actually able to respond to that interaction. A "fast" FID from a user's perspective would be immediate feedback from their actions on a site rather than a stalled experience. Any delay is a pain point and could correlate with interference from other aspects of the site loading when the user tries to interact with the site.

FID is very fast for the average CMS website on desktop—only Wix scores lower than 100%—and mixed on mobile. Most CMSs deliver mobile FID on an average site within a reasonable range of the desktop score. For Wix the number of websites that have a good FID on mobile is nearly half the desktop total.

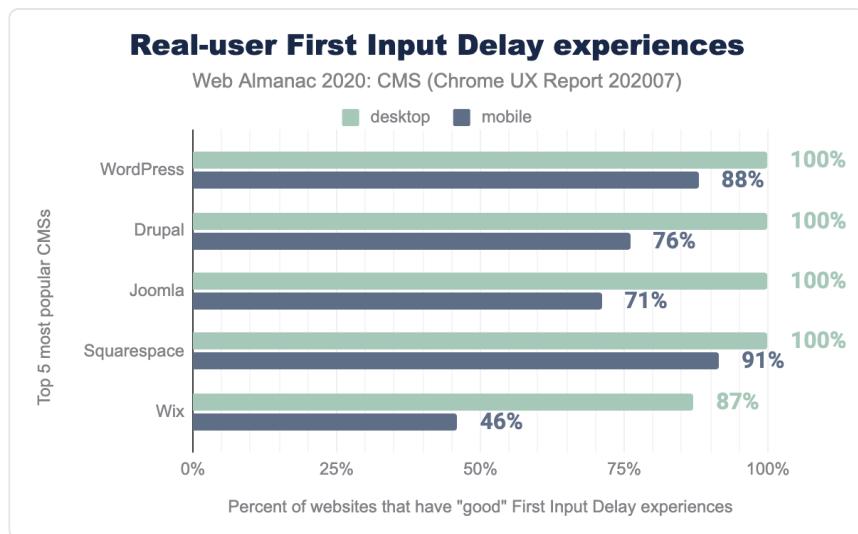


Figure 15.9. Real-user First Input Delay experiences.

The FID scores are generally good here, in contrast to the LCP scores. As suggested, the weight of individual pages on CMSs in addition to mobile connection quality or the lower performance of mobile devices relative to desktop, could play a role in the performance gaps that we see here affecting FID less.

There is a small margin of difference between the resources shipped to desktop and mobile versions of a website. Last year we noted that optimizing for the mobile experience was necessary. Average scores have increased on desktop and mobile, but further attention is required on mobile.

## Cumulative Layout Shift

Cumulative Layout Shift (CLS) measures the instability of content on a web page after the first 500ms of user input, and after the first user input. This is important on mobile in particular, where the user will tap where they want to take an action—such as a search bar—only for the location to move as additional images, ads, or similar load.

A score of 0.1 or below is measured as "good", over 0.25 is "poor", and anything in between is "needs improvement".

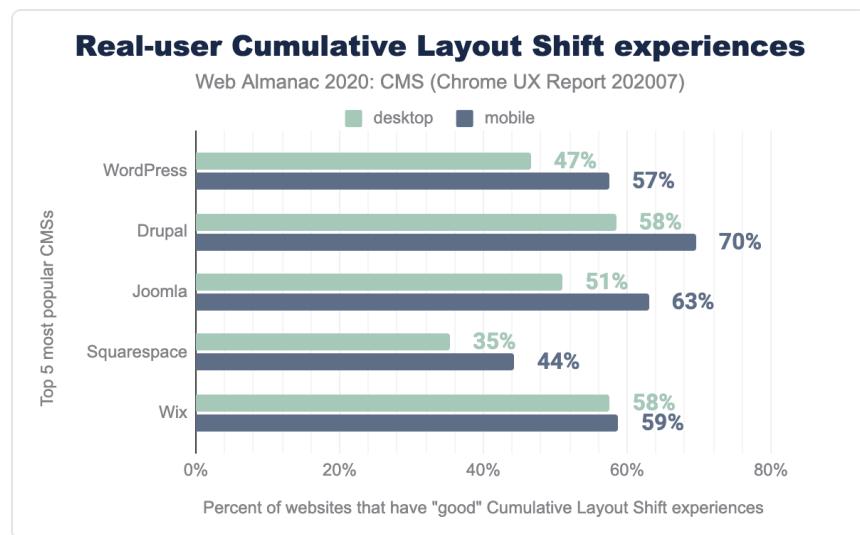


Figure 15.10. Real-user Cumulative Layout Shift experiences.

The top 5 CMSs could improve here. Only 50% of web pages loaded by a top 5 CMS have a "good" CLS experience, with this figure rising to 59% on mobile. Across all CMSs the average desktop score is 59% and average mobile score is 67%. This shows us all CMSs have work to do here, but the top 5 CMSs in particular need improvement.

## Lighthouse scores

Lighthouse is an open-source, automated tool designed to help developers assess and improve the quality of their websites. One key aspect of the tool is that it provides a set of audits to assess the status of a website in terms of performance, accessibility, SEO, progressive web apps, and more. For this year's chapter, we looked at two specific audit categories: SEO and accessibility.

### SEO

Search Engine Optimization (or SEO) is the practice of optimizing websites to make your website content more easily found in search engines. This is covered in more depth in our SEO chapter, but one part involves ensuring the site is coded in such a way to serve as much information to search engine crawlers to make it as easy as possible for them to show your site appropriately in search engine results. Compared to a custom created website, you would expect an CMS to provide good SEO capabilities, and the Lighthouse scores in this category show high marks:

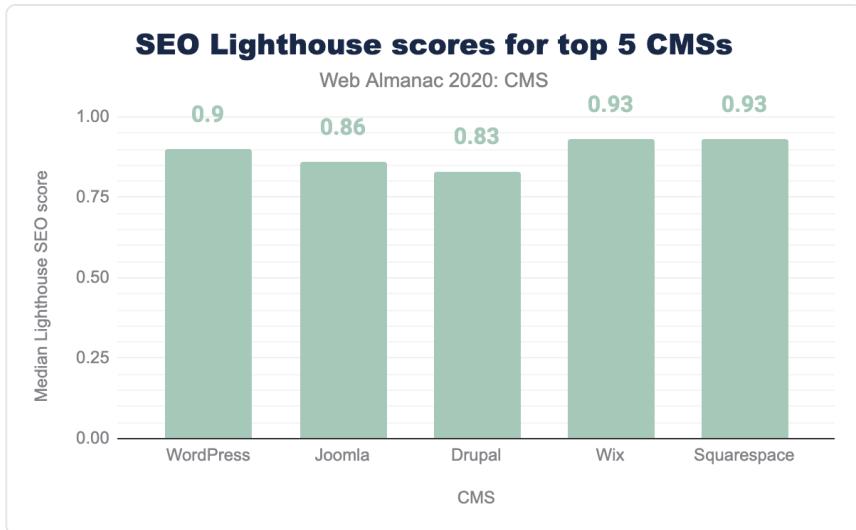


Figure 15.11. SEO Lighthouse scores for Top 5 CMSs.

All of the top 5 CMSs score highly here with median scores of 0.83 or above, with some reaching as high as 0.93. SEO can depend on the website owner making use of capabilities of a CMS but making those options easy to use in a CMS, and good defaults, can have big benefits for sites run on those CMSs.

## Accessibility

An accessible website is a site designed and developed so that people with disabilities can use them. Web accessibility also benefits people without disabilities, such as those on slow internet connections. A full discussion can be seen here, and in our Accessibility chapter.

Lighthouse provides a set of accessibility audits and it returns a weighted average of all of them (see Scoring Details for a full list of how each audit is weighted).

Each accessibility audit is either a pass or a fail, but unlike other Lighthouse audits, a page doesn't get points for partially passing an accessibility audit. For example, if some elements have screen reader-friendly names, but others don't, that page gets a 0 for the screen reader-friendly-names audit.

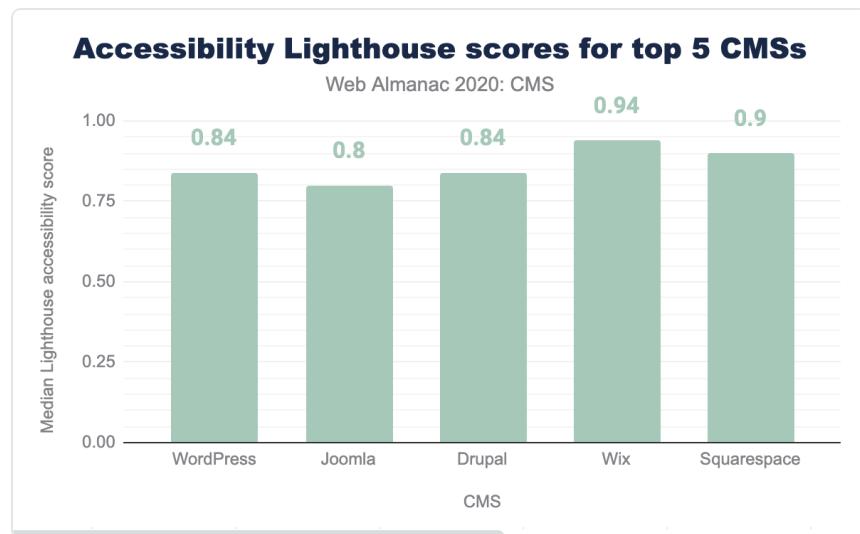


Figure 15.12. Accessibility Lighthouse scores for Top 5 CMSs.

The median Lighthouse accessibility score for the top 5 CMSs is all above 0.80. Across all CMSs, the average median Lighthouse score is 0.78, with a minimum of 0.44 and a maximum of 0.98. We thus see that the top 5 CMSs are better than average, with some better than others. Wix and Squarespace have the highest scores of the top 5. Possibly these platforms being proprietary helps here, as they're able to control the sites which are created more closely.

The bar should be higher here, though. An average score of 0.78 across all CMSs still leaves significant room for improvement, and the maximum score of 0.98 shows even the "best" CMS for accessibility compliance has room for improvement. Improving accessibility is essential and urgent work.

## Environmental impact

This year we've sought to better understand the impact of CMSs on the environment. The information and communications technology (ICT) industry accounts for 2% of global carbon emissions, and data centers specifically account for 0.3% of global carbon emissions. This puts the ICT industry's carbon footprint equivalent to the aviation industry's emissions from fuel. We don't have data on the role of CMSs here, but with our research showing 42% of websites use a CMS, it is clear CMSs play an important role in the efficiency of websites and their impact on the environment.

Our research looked at the average CMS page weight in KB and mapped this to CO<sub>2</sub> emissions using logic from carbonapi. This generated the following results, split by desktop and mobile:

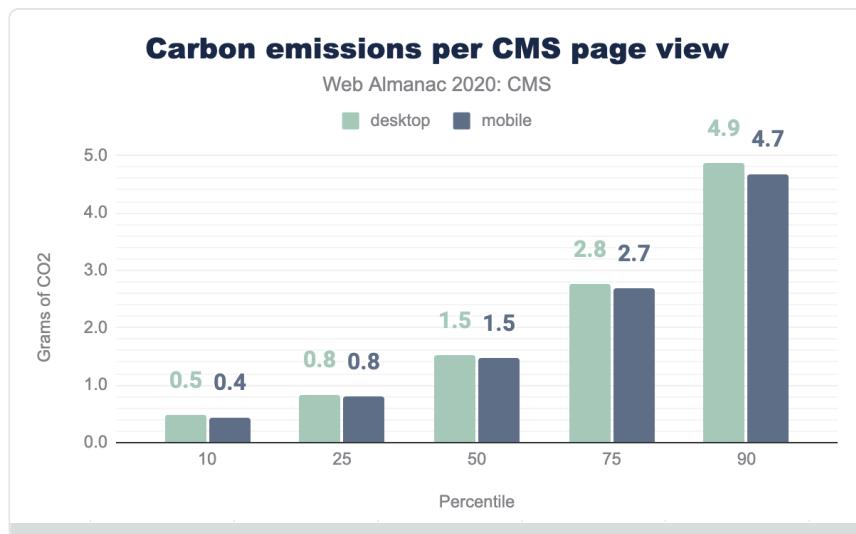


Figure 15.13. Carbon Emissions per CMS page view.

We found that the median CMS page load resulted in the transfer of 2.41 MB and thus the emission of 1.5g of CO<sub>2</sub>. This was the same for desktop and mobile. The most efficient percentile of CMS web pages result in the generation of at least one third less CO<sub>2</sub>, whilst the least efficient percentile of CMS web pages goes the other way: over one third less efficient than the median. The most efficient percentile of pages is approximately ten times more efficient than the least efficient percentile.

CMSs power every type of website, so this discrepancy is not surprising. CMSs can, however, influence at the platform-level the efficiency of websites they create.

Page weights are important here. The average desktop CMS web page loads 2.4 MB of HTML, CSS, JavaScript, media, etc. 10% of pages, however, load over 7 MB of this data. On mobile devices the average web page loads 0.1 MB fewer than on desktop, with at least this number being true across all percentiles:

## Distribution of CMS page sizes

Web Almanac 2020: CMS

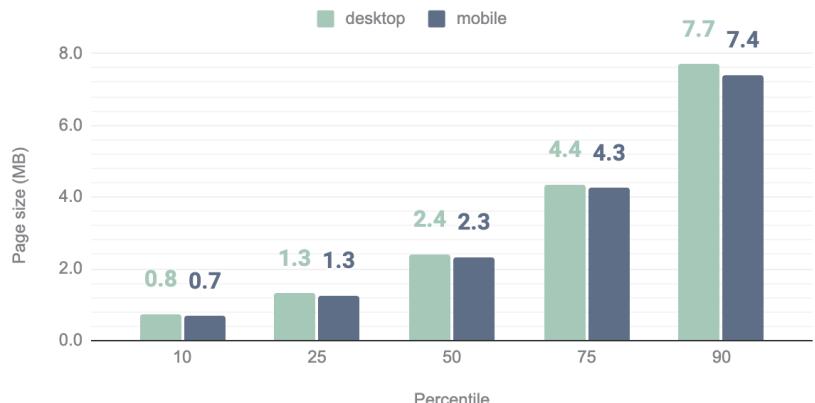


Figure 15.14. Distribution of CMS page sizes.

CMS often load third party resources, such as external images, videos, scripts, or stylesheets:

## Third party bytes

Web Almanac 2020: CMS

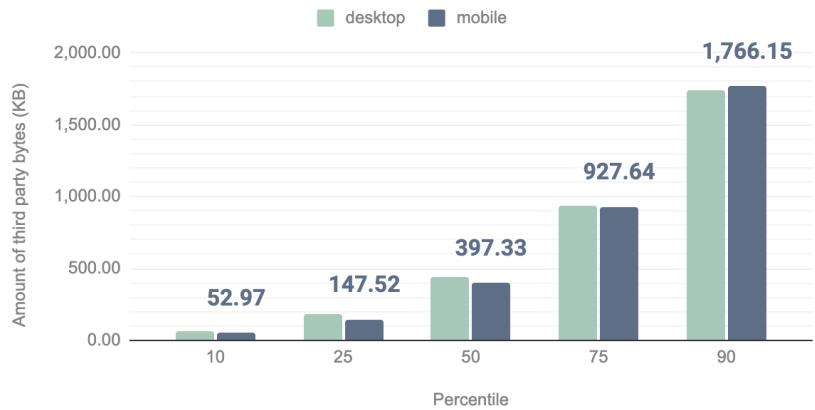


Figure 15.15. Third party bytes.

We find that the median desktop CMS page has 27 third-party requests with 436 KB of content, with the mobile equivalent generating 26 requests with 397 KB of content.

One of the main ways a CMS can influence its page load size is by supporting and encouraging the usage of more efficient formats. Images are behind only video in their contribution to page weight.

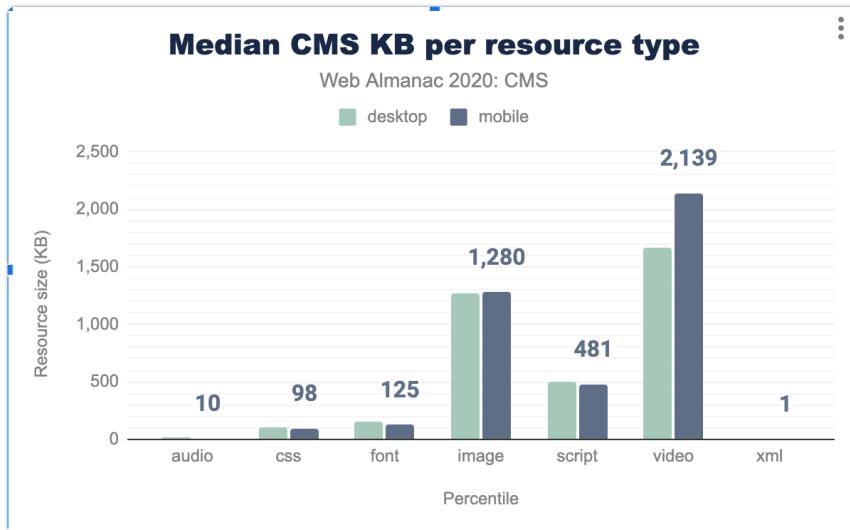
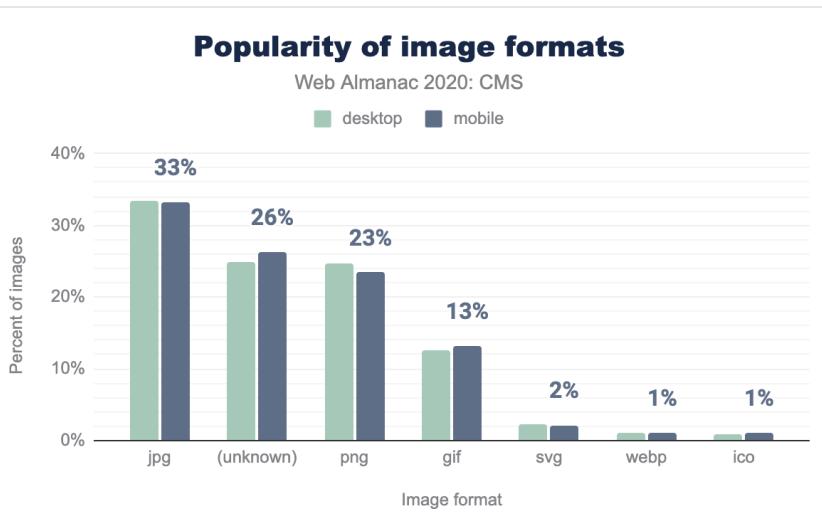


Figure 15.16. Median CMS KB per resource type.

Video contributes a larger percentage per resource type here. Making video more efficient, or other mechanisms such as the impact of stopping autoplay, are interesting areas for future research. Here our focus is on images. Popular image formats are JPEG, PNG, GIF, SVG, WebP, and ICO. Of these, WebP is the most efficient in most situations, with WebP lossless images 26% smaller than equivalent PNGs and 25-34% smaller than comparable JPGs. We see, however, that WebP is the second least popular image format across all CMS pages:



*Figure 15.17. Popularity of image formats.*

Of the top 5 CMSs, only Wix automatically converts and serves images in the WebP format. WordPress, Drupal, and Joomla support WebP with extensions, whilst at the time of writing Squarespace does not support WebP.

As we saw earlier, Wix had the lowest proportion of sites with a "good" LCP. While we know that Wix is making efficient use of image bytes in WebP, there are clearly other issues affecting its LCP performance beyond image formats that we aren't controlling for here. WebP is, however, a more efficient format and improved native support for the format by the most popular CMSs would be beneficial.

Image formats are one mechanism for making images more efficient. Other mechanisms such as "lazy loading" images would benefit from future research.

We're unable to fully answer the question of the impact of CMSs on the environment, but we are contributing to an answer. CMSs have a responsibility to take environmental impact seriously and decreasing the average page weight is important work.

## Conclusion

CMSs have only gotten more important in the last year. They are essential for how content is created and consumed on the internet, and there are no signs that this will change in the foreseeable future. CMSs are set to become more important with each year passing.

We have reviewed the adoption of CMSs, user experience of websites created by these CMSs, and for the first time looked at the impact of CMSs on the environment. We have answered many questions here but leave further questions unanswered. Further research building on this chapter will be gratefully received. We have also highlighted some areas which need attention by the CMSs. We hope there will be progress to share in the 2021 report.

CMSs are vital for the success of the internet and open web. Let's work towards continued progress.

## Author



### Alex Denning

🐦 @AlexDenning    🌐 alexdenning    🌐 <https://getellipsis.com>

Alex Denning is the Founder of Ellipsis Marketing<sup>30</sup>, a marketing agency for WordPress businesses. Alex is a WordPress Core Contributor and has helped organize WordCamp London<sup>31</sup>.

---

30. <https://getellipsis.com/>  
31. <https://london.wordcamp.org/>



# Part III Chapter 16

# Ecommerce [UNEDITED]



Written by Rockey Nebhwani

Reviewed by Jason Haralson and Drewz

Analyzed by Jason Haralson

## Author



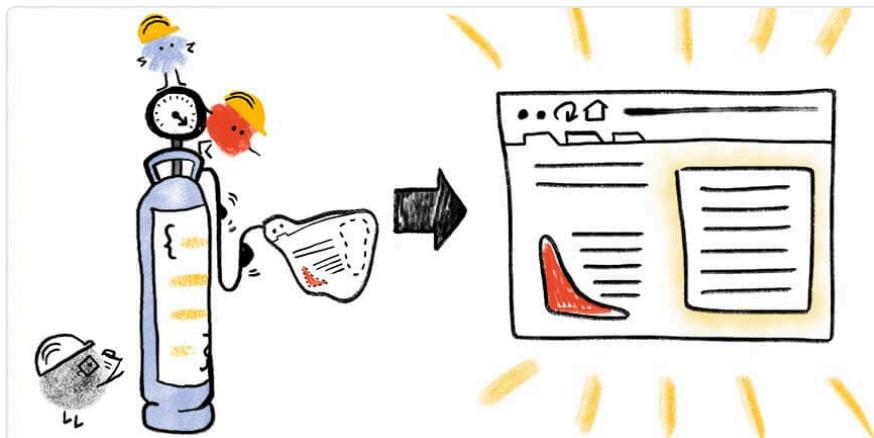
Rockey Nebhwani

[Twitter](#) @rnebwani [GitHub](#) rockeynebwani [LinkedIn](#) rockeynebwani



# Part III Chapter 17

# Jamstack



*Written by Ahmad Awais*

*Reviewed by Maedah Batool and Nicolas Goutay*

*Analyzed by Artem Denysov and Brian Rinaldi*

## Introduction

Jamstack is a relatively new concept of an architecture designed to make the web faster, more secure, and easier to scale. It builds on many of the tools and workflows which developers love, and which maximizes productivity.

The core principles of Jamstack are pre-rendering your site pages and decoupling the frontend from the backend. It relies on the idea of delivering the frontend content hosted separately on a CDN provider that uses APIs (for example, a headless CMS) as its backend if any.

The HTTP Archive crawls millions of pages every month and runs them through a private instance of WebPageTest to store key information on every page crawled. You can learn more about this in our methodology page. In the context of Jamstack, HTTP Archive provides extensive information on the usage of the frameworks and CDNs for the entire web. This chapter consolidates and analyzes many of these trends.

The goals of this chapter are to estimate and analyze the growth of the Jamstack sites, the

performance of popular Jamstack frameworks, as well as an analysis of real user experience using the Core Web Vitals metrics.

*It should be noted that our analysis is limited by those Jamstacks that make themselves easily identifiable using Wappalyzer. This means our data does not include some popular Jamstacks like Eleventy which make a deliberate choice to not make themselves identifiable. While we would ideally include all Jamstacks, we believe there is still plenty of value in analyzing the significant data we do have.*

## Adoption of Jamstack

Our analysis throughout this work looks at desktop and mobile websites. The vast majority of URLs we looked at are in both datasets, but some URLs are only accessed by desktop or mobile devices. This can cause small divergences in the data, and we thus look at desktop and mobile results separately.

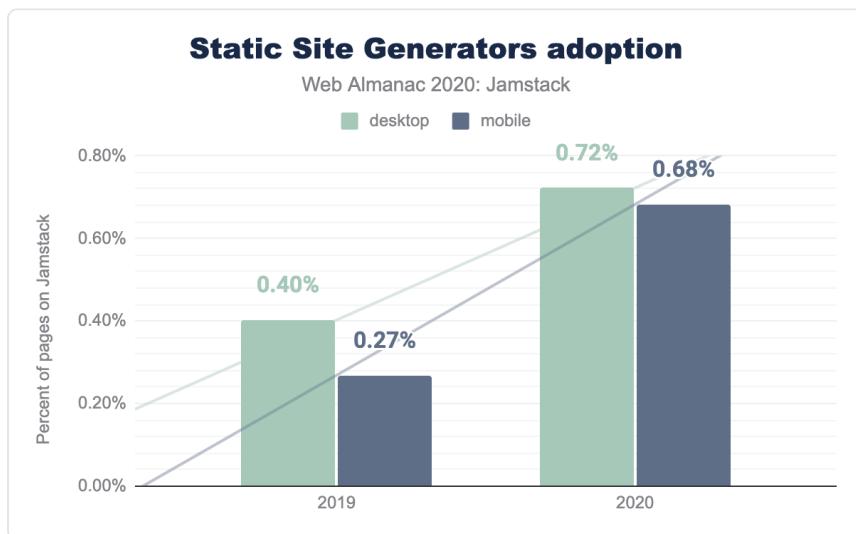


Figure 17.1. Jamstack adoption trend.

More than 0.7% of web pages are powered by Jamstack and breaks down to 0.72% on desktop, up from 0.40% in 2019, and 0.68% on mobile, up from 0.27% in 2019.

<b>Year</b>	<b>Desktop</b>	<b>Mobile</b>
2019	0.40%	0.27%
2020	0.72%	0.68%
<b>% Change</b>	<b>79%</b>	<b>154%</b>

Figure 17.2. Jamstack adoption statistics.

The increase in desktop web pages powered by a Jamstack framework is 79% from last year. On mobile, this increase is almost two folds, at 154%. This is a significant growth from 2019, especially for mobile pages. We believe this is a sign of the steady growth of the Jamstack community.

## Jamstack frameworks

Our analysis counted 12 separate Jamstack frameworks. Only five frameworks had more than 1% share: Next.js, Gatsby, Hugo, Jekyll are the top contenders for the Jamstack market share.

In 2020, most of the Jamstack market share seems distributed between the top four frameworks. Interestingly, Next.js has 72.5% usage share. This is nearly five times the share of the next most popular Jamstack framework, Gatsby at just under 15%!

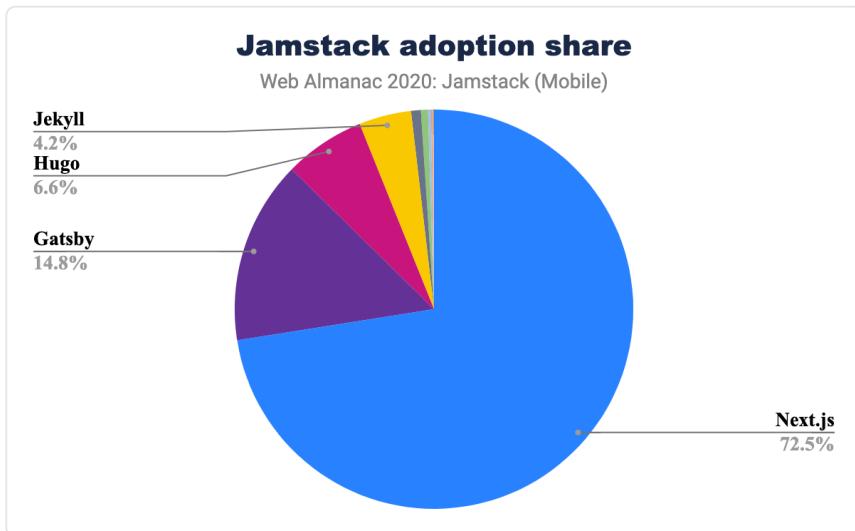


Figure 17.3. Jamstack adoption share pie chart 2020.

## Framework adoption changes

Looking at the year on year growth, we see that Next.js has increased its lead over its competitors in the last year:

Jamstack	2019	2020	% change
Next.js	61.06%	72.51%	19%
Gatsby	15.87%	14.84%	-6%
Hugo	12.12%	6.56%	-46%
Jekyll	7.92%	4.24%	-46%
Hexo	1.48%	0.79%	-46%
Gridsome	0.25%	0.57%	133%
Octopress	0.78%	0.24%	-69%
Pelican	0.39%	0.14%	-65%
VuePress		0.06%	
Phenomic	0.13%	0.03%	-78%
Saber		0.01%	
Cecil		0.01%	

Figure 17.4. Relative % adoption of Jamstack frameworks

And concentrating on the top 5 Jamstacks further shows Next.js's lead:

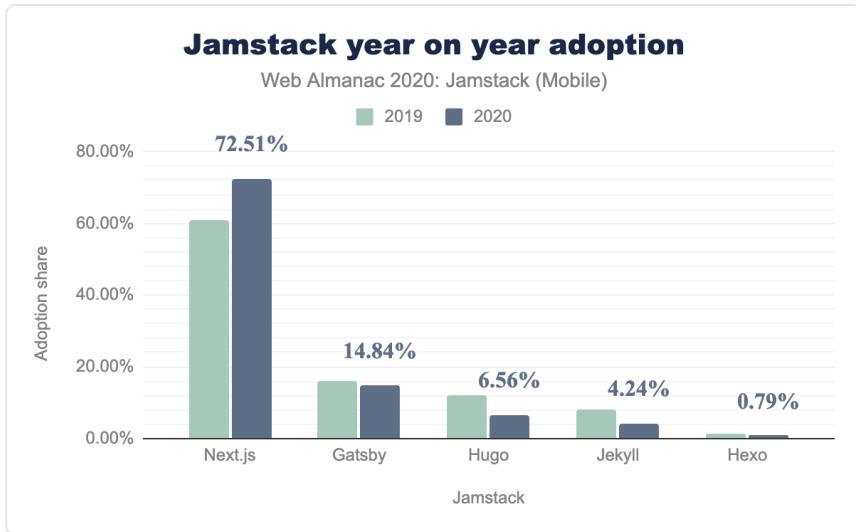


Figure 17.5. Jamstack adoption share year on year.

It's worth noting here the fact that Next.js websites include a mix of both Static Site Generated (SSG) pages and Server-Side Rendered (SSR) pages. This is due to the lack of our ability to measure them separately. This means that the analysis may include sites that are mostly or partially server-rendered, meaning they do not fall under the traditional definition of a Jamstack site. Nonetheless, it appears that this hybrid nature of Next.js gives it a competitive advantage over other frameworks hence making it more popular.

## Environmental impact

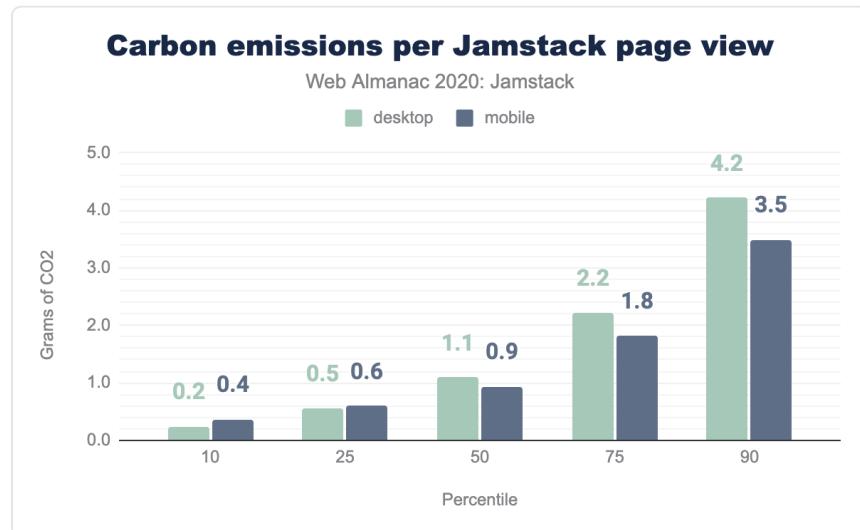
This year we have sought to better understand the impact of Jamstack sites on the environment. The information and communications technology (ICT) industry accounts for 2% of global carbon emissions, and data centers specifically account for 0.3% of global carbon emissions. This puts the ICT industry's carbon footprint equivalent to the aviation industry's emissions from fuel.

Jamstack is often credited for being mindful of performance. In the next section, we look into the carbon emissions of Jamstack websites.

## Page weight

Our research looked at the average Jamstack page weight in KB and mapped this to CO<sub>2</sub> emissions using logic from the Carbon API. This generated the following results, split by

desktop and mobile:



*Figure 17.6. Carbon Emissions per Jamstack page view.*

We found that the median Jamstack page load resulted in the transfer of 1.5 MB of various assets and thus the emission of 1.1 grams of CO<sub>2</sub> for desktop and 0.9 grams for mobile. This was the same for desktop and mobile. The most efficient percentile of Jamstack web pages result in the generation of at least one third less CO<sub>2</sub> than the median, whilst the least efficient percentile of Jamstack web pages goes the other way, generating around four times more.

Page weights are important here. The average desktop Jamstack web page loads 1.5 MB of video, image, script, font, CSS, and audio data. 10% of pages, however, load over 4 MB of this data. On mobile devices, the average web page loads 0.7 MB fewer than on desktop, a fact consistent across all percentiles.

## Image formats

Popular image formats are PNG, JPG, GIF, SVG, WebP, and ICO. Of these, WebP is the most efficient in most situations, with WebP lossless images 26% smaller than equivalent PNGs and 25-34% smaller than comparable JPGs. We see, however, that WebP is the second least popular image format across all Jamstack pages, where PNG is the most popular both for mobile and desktop. Only slightly less popular is JPG whereas GIF is almost 20% of all the images used on Jamstack sites. An interesting discovery is SVG which is almost twice as popular on mobile sites as desktop sites.

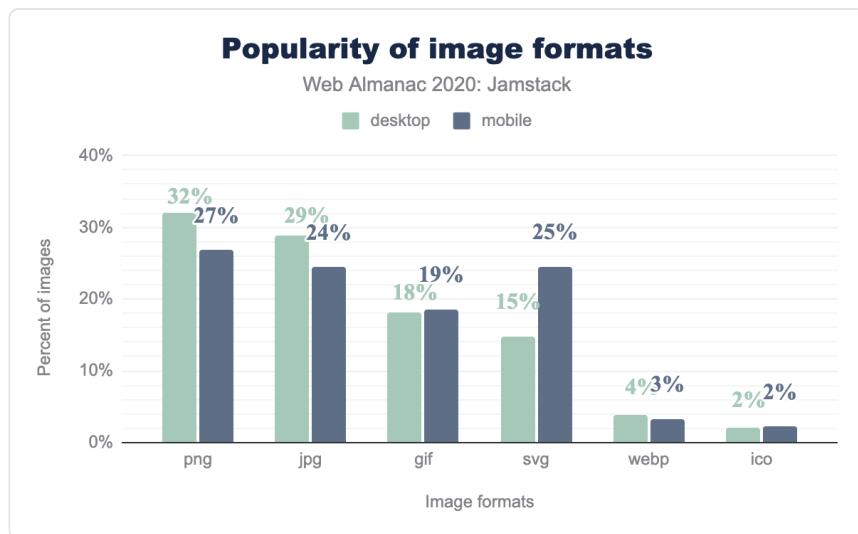
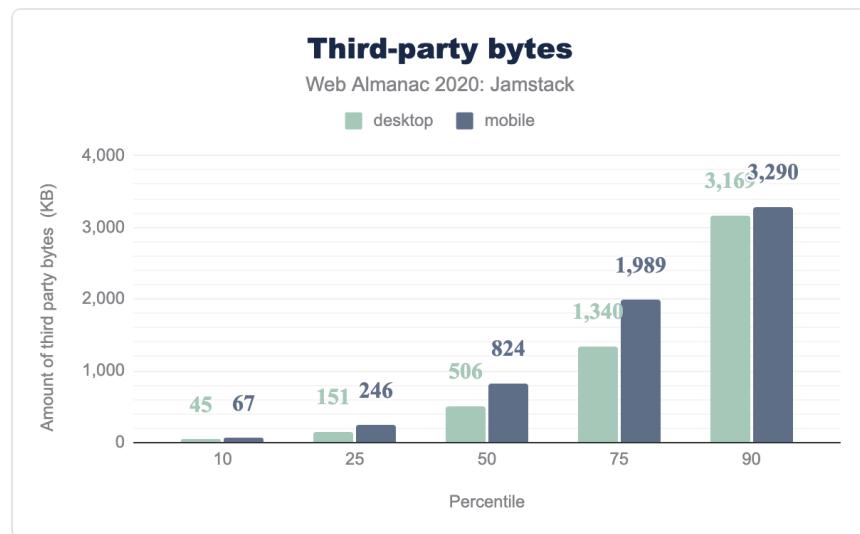


Figure 17.7. Popularity of image formats.

## Third-party bytes

Jamstack sites, like most websites, often load third-party resources, such as external images, videos, scripts, or stylesheets:



*Figure 17.8. Third party bytes.*

We find that the median desktop Jamstack page has 28 third-party requests with 506 KB of content, with the mobile equivalent generating 46 requests with 824 KB of content. Whereas 10% of the desktop sites have 132 requests with 3.1MB of content which is only superseded by 168 requests on mobile with 3.2MB of content.

## User experience

Jamstack websites are often said to offer a good user experience. It's what the entire concept of separating the frontend from the backend and hosting it on the CDN edge is all about. We aim to shed light on real-world user experience when using Jamstack websites using the recently launched Core Web Vitals.

The Core Web Vitals are three important factors which can shed light on our understanding of how users are experiencing Jamstack pages in the wild:

- Largest Contentful Paint (LCP)
- First Input Delay (FID)
- Cumulative Layout Shift (CLS)

These metrics aim to cover the core elements which are indicative of a great web user experience. Let's we take a look at the real-world Core Web Vitals statistics of the top-five Jamstack frameworks.

## Largest Contentful Paint

Largest Contentful Paint (LCP) measures the point when the page's main content has likely loaded and thus the page is useful to the user. It does this by measuring the render time of the largest image or text block visible within the viewport.

This is different from First Contentful Paint (FCP), which measures from page load until content such as text or an image is first displayed. LCP is regarded as a good proxy for measuring when the main content of a page is loaded.

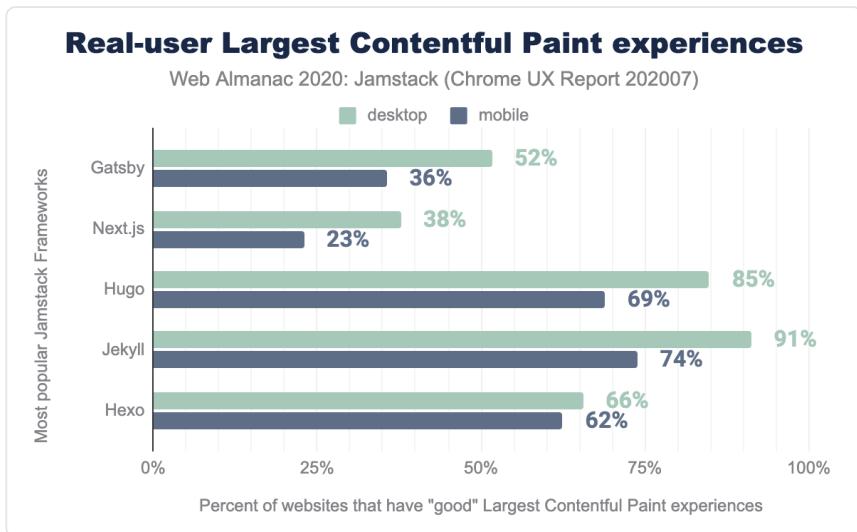


Figure 17.9. Real-user Largest Contentful Paint experiences.

A "good" LCP is regarded as under 2.5 seconds. Hugo, Jekyll, and Hexo have impressive LCP scores all above 50% with Jekyll and Hugo on desktop at 91% and 85% on desktop respectively. Gatsby and Next.js sites lagged – scoring 52% and 38% respectively on desktop, and 36% and 23% on mobile.

This might be attributed to the fact that most of the sites built with Next.js and Gatsby have complex layouts and high page weights, in comparison with Hugo and Jekyll which are primarily used to produce static content sites with fewer or no dynamic parts. For what it's worth, you don't have to use React or any other JavaScript framework with Hugo or Jekyll.

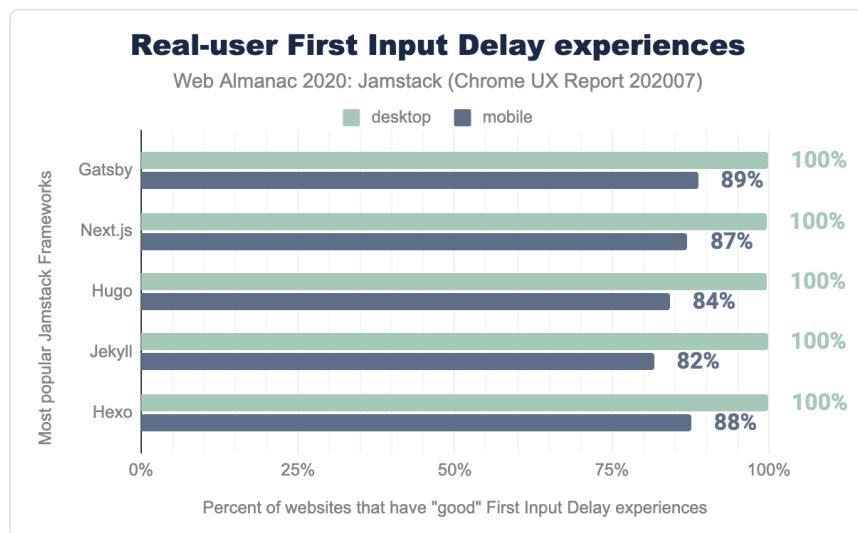
As we explored in the section above, high page weights can have a possible impact on the environment. However, this also affects LCP performance, which is either very good or generally bad depending on the Jamstack framework. This can have an impact on the real user experience as well.

## First Input Delay

First Input Delay (FID) measures the time from when a user first interacts with your site (i.e. when they click a link, tap on a button, or use a custom, JavaScript-powered control) to the time when the browser is actually able to respond to that interaction.

A "fast" FID from a user's perspective would provide immediate feedback from their actions on a site rather than a stalled experience. This delay is a pain point and could correlate with interference from other aspects of the site loading when the user tries to interact with the site.

FID is extremely fast for the average Jamstack website on desktop – most popular frameworks score 100% – and above 80% on mobile.



*Figure 17.10. Real-user First Input Delay experiences.*

There is a small margin of difference between the resources shipped to desktop and mobile versions of a website. The FID scores are generally very good here, but it is interesting this does not translate to similar LCP scores. As suggested, the weight of individual pages on Jamstack sites in addition to mobile connection quality could play a role in the performance gaps that we see here.

## Cumulative Layout Shift

Cumulative Layout Shift (CLS) measures the instability of content on a web page within the first 500ms of user input. CLS measures any layout changes which happen after user input. This is important on mobile in particular, where the user will tap where they want to take an action –

such as a search bar – only for the location to move as additional images, ads, or similar load.

A score of 0.1 or below is good, over 0.25 is poor, and anything in between needs improvement.

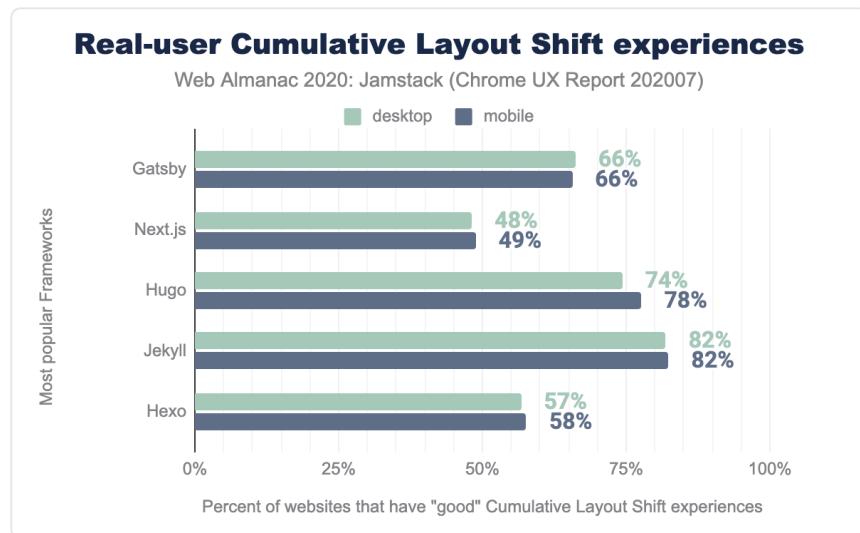


Figure 17.11. Real-user Cumulative Layout Shift experiences.

The top-five Jamstack frameworks do OK here. About 65% of web pages loaded by top-five Jamstack frameworks have a "good" CLS experience, with this figure rising to 82% on mobile. Across all the average desktop and mobile score is 65%. Next.js is barely reaching 50% which is the lowest of all and has work to do here. Educating developers and documenting how to avoid bad CLS scores can go a long way.

## Conclusion

Jamstack, both as a concept and a stack, has picked up importance in the last year. Stats suggest almost twice as many Jamstack sites exist now than in 2019. Developers enjoy a better development experience by separating the frontend from the backend (a headless CMS, serverless functions, or third-party services). But what about the real-user experience of browsing Jamstack sites?

We've reviewed the adoption of Jamstack, user experience of websites created by these Jamstack frameworks, and for the first time looked at the impact of Jamstack on the environment. We have answered many questions here but leave further questions unanswered.

There are frameworks like Eleventy which we weren't able to measure or analyze since there is

no pattern available to determine the usage of such frameworks, which has an impact on the data presented here. Next.js dominates usage and offers both Static Site Generation and Server-Side Rendering, separating the two in this data is nearly impossible since it also offers incremental Static Generation. Further research building on this chapter will be gratefully received.

Moreover, we have highlighted some areas which need attention from the Jamstack community. We hope there will be progress to share in the 2021 report. Different Jamstack frameworks can start to document how to improve real user experience by looking at Core Web Vitals.

Vercel, one of the CDNs meant to host Jamstack sites, has built an analytics offering called Real User Experience Score. While other performance measuring tools like Lighthouse estimate your user's experience by running a simulation in a lab, Vercel's Real Experience Score is calculated using real data points collected from the devices of the actual users of your application.

It is probably worth noting here that Vercel created and maintains Next.js, since Next.js had low LCP scores. This new offering could mean we can hope to see a marked improvement in those scores next year. This would be extremely helpful information for users and developers alike.

Jamstack frameworks are improving the developer experience of building sites. Let's work towards continued progress for improving the real-user experience of browsing Jamstack sites.

## Author



### Ahmad Awais

 @MrAhmadAwais  ahmadawais  <https://AhmadAwais.com>

Ahmad Awais is an award-winning open-source engineer, Google Developers Expert Dev Advocate, Node.js Community Committee Outreach Lead, WordPress Core Dev, and VP of Engineering DevRel at WGA. He has authored various open-source software tools used by millions of developers worldwide. Like his Shades of Purple<sup>32</sup> code-theme or projects like the corona-cli<sup>33</sup>. Awais loves to teach. Over 20,000 developers are learning from his courses<sup>34</sup> i.e. Node CLI<sup>35</sup>, VSCode.pro<sup>36</sup>, and Next.js Beginner<sup>37</sup>. Awais received FOSS community leadership recognition as one of the 12 featured GitHub Stars<sup>38</sup>. He is a member of the Smashing Magazine Experts Panel; featured & published author at CSS-Tricks, Tuts+, Scotch.io, SitePoint. You can mostly find him on Twitter @MrAhmadAwais where he tweets his #OneDevMinute<sup>39</sup> developer tips.

32. <https://shadesofpurple.pro/more>

33. <https://github.com/AhmadAwais/corona-cli>

- 
34. <https://AhmadAwais.com/courses>
  35. <https://nodecli.com/>
  36. <https://vscode.pro/>
  37. <https://nextjsbeginner.com/>
  38. <https://ahmadawais.com/github-stars/>
  39. <https://awais.dev/odint>



# Part IV Chapter 18

# Page Weight [UNEDITED]



Written by Henri Helvetica

Reviewed by Tammy Everts, Boris Schapira, and Katie Hempenius

Analyzed by Paul Calvano

## Author



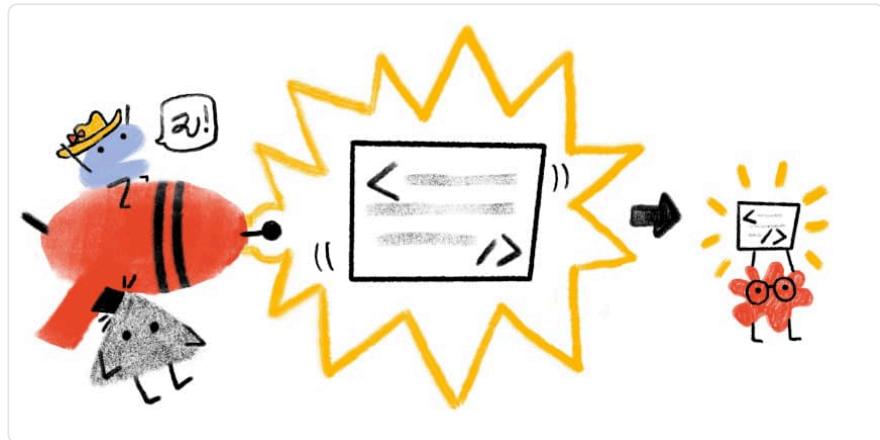
Henri Helvetica

[@HenriHelvetica](#) [henrihelvetica](#)



# Part IV Chapter 19

# Compression [UNEDITED]



Written by Moritz Firsching, Luca Versari, Sami Boukortt, and Jyrki Alakuijala

Reviewed by Paul Calvano

Analyzed by Abby Tsai

## Introduction

Using HTTP compression makes a website load faster and therefore guarantees a better user experience. Running no compression on HTTP makes for a worse user experience, may affect the growth rate of the related web service and affects search rankings. Using compression likely produces a web experience that performs better on metrics such as faster Largest Contentful Paint. Using compression reduces page weight, improves web performance, and therefore is an important part of search engine optimization.

While lossy compression is often acceptable for images and other media types, for text we want to use lossless compression, i.e. recover the exact text after decompression.

## What type of content should we compress?

For most text-based assets, such as HTML, CSS, JavaScript, JSON or SVG, as well as certain

non-text formats such as woff, ttf, ico, using compression is recommended. Here is an overview over what compression methods are currently used for different content types:

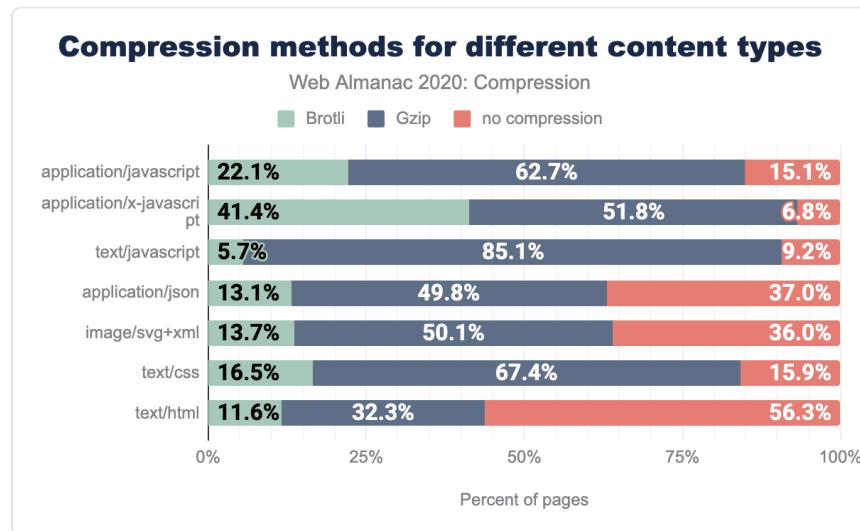


Figure 19.1. Compression Methods for Different Content Types

The figure shows the percentages of the request of a certain content type using either brotli, gzip or no text compression. It is surprising that while all those content types would profit from compression, the range of percentages varies widely over the different content types: only 44% use compression for `text/html` against 93% `application/x-javascript`.

## How to use HTTP compression?

To reduce the file sizes of the files that we plan to serve you could first use some minimizers, e.g. `HTMLMinifier`, `CSSNano` or `UglifyJS`. However bigger gains are expected from using compression.

There are two ways of doing the compression on the server side:

- Precompressed (compress and save assets ahead of time)
- Dynamically Compressed (compress assets on-the-fly after a request is made)

Since precompression is done beforehand, we can spend more time compressing the assets. For dynamically compressed resources we need to choose the compression levels such that compression takes less time than the difference between sending an uncompressed versus a compressed file. Currently practically all text compression is done by one of two HTTP content encodings: Gzip and brotli. Both are widely supported by browsers: can I use brotli/can I use

**gzip** When you want to use gzip, consider using Zopfli, which generates smaller gzip compatible files. This should be done especially for precompressed resources, since here the greatest gains are expected. See this comparison between gzip and zopfli that takes into account different compression levels for gzip.

Many popular servers support dynamically and/or pre-compressed HTTP ([https://en.wikipedia.org/wiki/HTTP\\_compression#Servers\\_that\\_support\\_HTTP\\_compression](https://en.wikipedia.org/wiki/HTTP_compression#Servers_that_support_HTTP_compression)) and many of them support Brotli: <https://en.wikipedia.org/wiki/Brotli>

Here are some general recommendations on what compression levels to use:

	<b>brotli</b>	<b>gzip</b>
precompressed	11	9 or zopfli
dynamically compressed	5	6

Figure 19.2. Recommended compression levels to use.

Currently, when compression is used, the split between brotli and gzip is about 23% / 77%.

Approximately 60% of HTTP responses are delivered with text-based compression. This may seem like a surprising statistic, but keep in mind that it is based on all HTTP requests in the dataset. Some content, such as images, will not benefit from these compression algorithms. The table below summarizes the percentage of requests served with each content encoding.

<b>Percent of Requests</b>			
<b>Content Encoding</b>	<b>Desktop</b>	<b>Mobile</b>	<b>Combined</b>
No Text Compression	60.06%	59.31%	59.67%
gzip	30.82%	31.56%	31.21%
br	9.10%	9.11%	9.11%
Other	0.02%	0.02%	0.02%

Figure 19.3. Adoption of compression algorithms.

Of the resources that are served compressed, the majority are using either gzip (77%) or brotli (23%). The other compression algorithms are used infrequently.

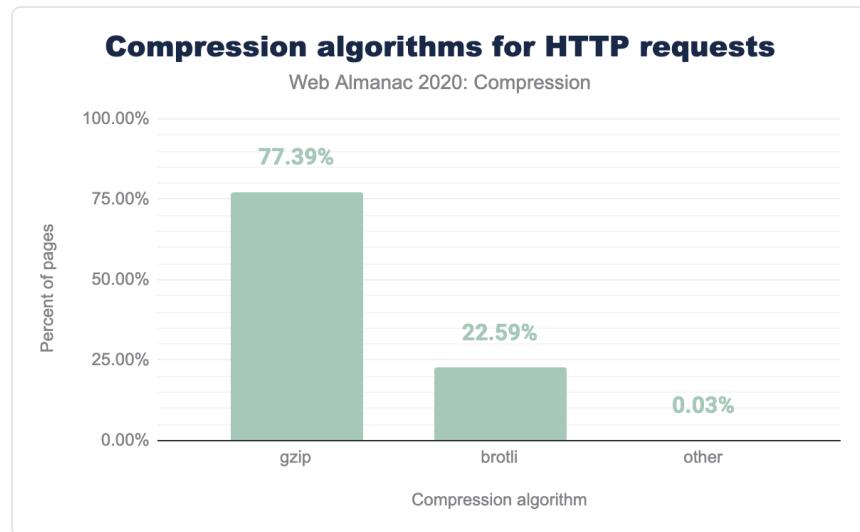


Figure 19.4. Compression algorithms.

## Current state of HTTP compression

In the graph below, the top 11 content types are displayed with box sizes representing the relative number of requests. The color of each box represents how many of these resources were served compressed. Most of the media content is shaded orange, which is expected since gzip and brotli would have little to no benefit for them. Most of the text content is shaded blue to indicate that they are being compressed. However, the light blue shading for some content types indicate that they are not compressed as consistently as the others.

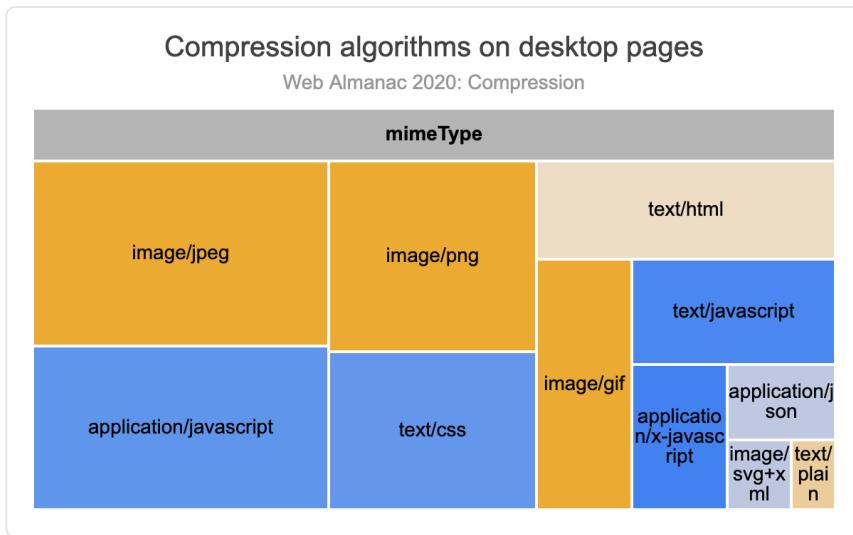


Figure 19.5. Top compressed content types.

Figure 19.1 above breaks down the percentages indicated as shadings in Figure 19.4 for the data types that should use compression. They are almost identical for desktop and mobile. Here's the analogous figure for those data types that ordinarily don't profit from further compression:

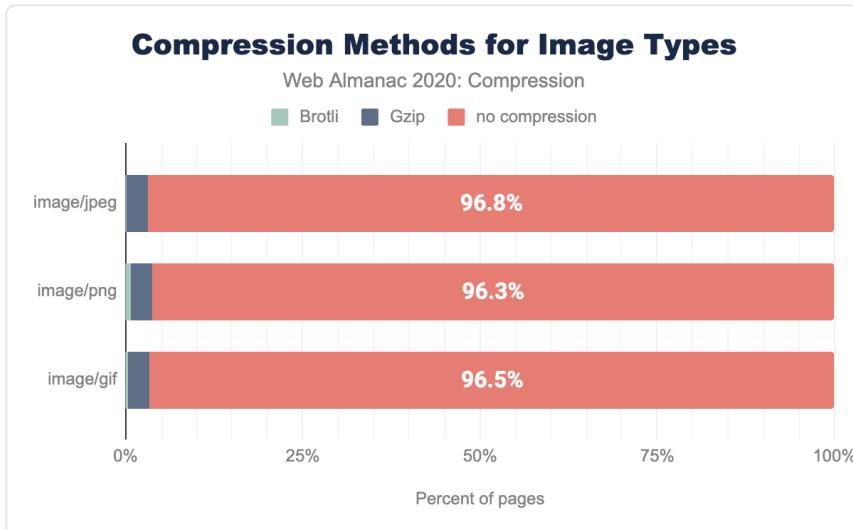


Figure 19.6. Compression by content type as a percent for desktop.

## First-party vs third-party compression

In the Third Parties chapter, we learn about third parties and their impact on performance. When we compare compression techniques between first and third parties, we can see that third-party content tends to be compressed more than first-party content.

Additionally, the percentage of brotli compression is higher for third-party content. This is likely due to the number of resources served from the larger third parties that typically support brotli, such as Google and Facebook.

<b>Content Encoding</b>	<b>Desktop</b>		<b>Mobile</b>	
	<b>First-Party</b>	<b>Third-Party</b>	<b>First-Party</b>	<b>Third-Party</b>
No Text Compression	61.93%	57.81%	60.36%	58.11%
gzip	30.95%	30.66%	32.36%	30.65%
br	7.09%	11.51%	7.26%	11.22%
deflate	0.02%	0.01%	0.02%	0.01%
Other / Invalid	0.01%	0.01%	0.01%	0.01%

Figure 19.7. First-party versus third-party compression by device type.

Comparing with last year's results, we can see that there was a significant increase in the use of compression, notably brotli for first parties, almost to the point that the use of compression is around 40% for both first and third party and for desktop and mobile. However within the requests that do use compression, for first party the ratio of brotli compression is only 18%, while the ratio for third party is 27%.

## How to Analyze compression on your sites

You can use Firefox Developer Tools or Chrome DevTools to quickly figure out for what content a website already uses some kind of compression. For this go to Network tab, right click and activate "Content Encoding" under Response Headers. Hovering over the size of individual files you will see "transferred over network" and "resource size". Aggregated for the entire site one can see size/transferred size for Firefox and "transferred" and "resources" for Chrome on the bottom left hand side of the Network tab.

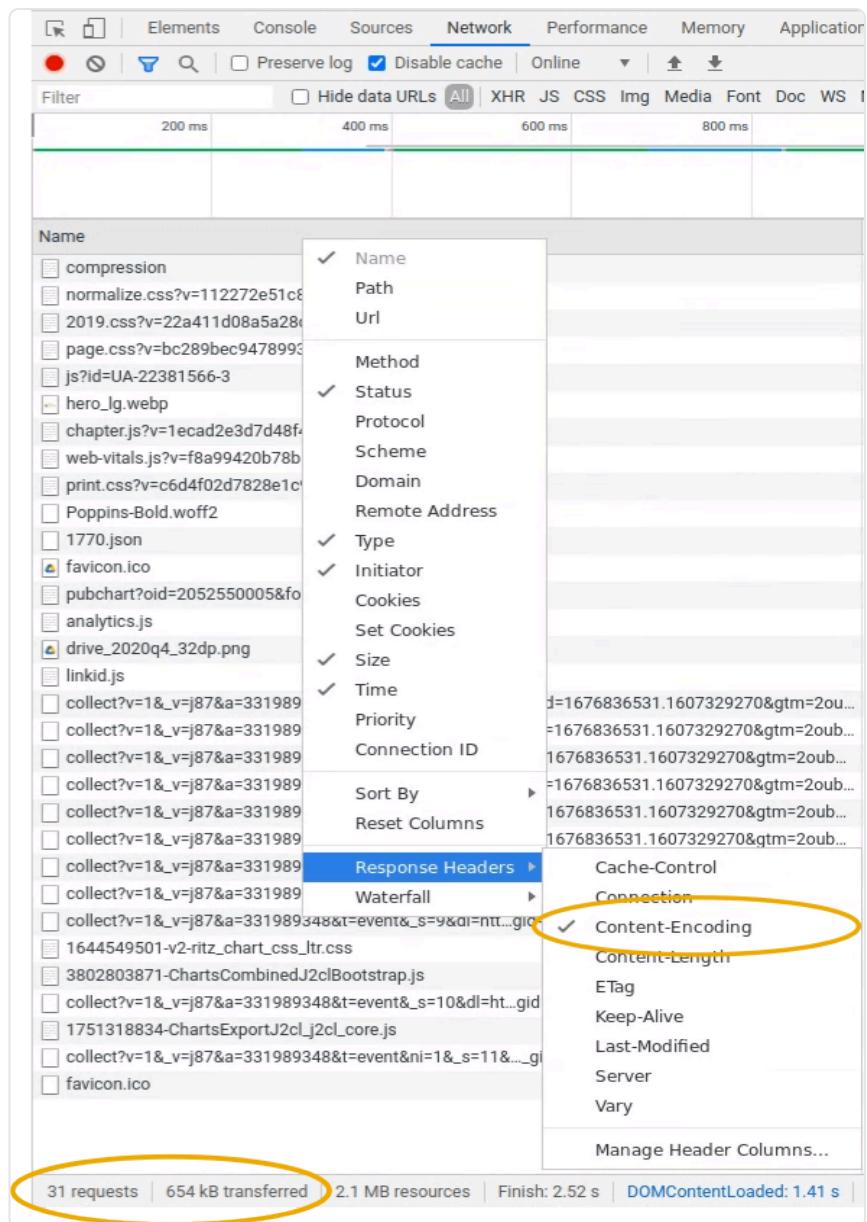


Figure 19.8. Use DevTools to check if content encoding is used on your site

Another tool to better understand compression on your site is Google's Lighthouse tool enables users to run a series of audits against web pages. The text compression audit evaluates whether

a site can benefit from additional text-based compression. It does this by attempting to compress resources and evaluate whether an object's size can be reduced by at least 10% and 1,400 bytes. Depending on the score, you may see a compression recommendation in the results, with a list of specific resources that could be compressed.

Because the HTTP Archive runs Lighthouse audits for each mobile page, we can aggregate the scores across all sites to learn how much opportunity there is to compress more content. Overall, 74% of websites are passing this audit and almost 13% of websites have scored below a 40. Compared to last year's 62.5%, this year already 74% of the observed pages have the best text compression Lighthouse audio score.

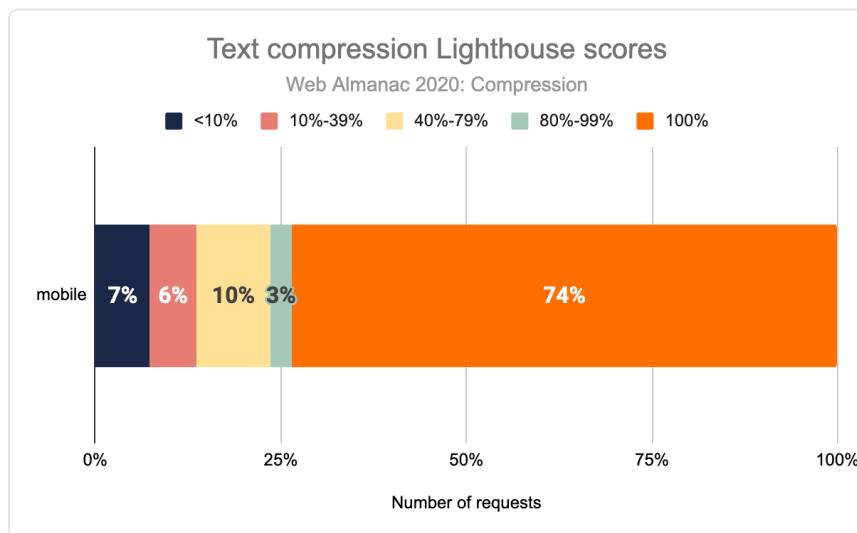


Figure 19.9. Lighthouse "enable text compression" audit scores.

## Conclusion

Compared with last year's almanac, there is a clear trend towards using more text compression. The number of requests that don't use any text compression went down a little more than 2%, while at the same time the use of brotli has increased by almost 2%. The Lighthouse scores have improved significantly.

Text compression is widely used for the relevant formats, although there is still a significant percentage of the http-requests that could benefit from additional compression. You can profit from taking a close look at the configuration of your server and set compression methods and levels to your need. A great impact for a more positive user experience could be made by carefully choosing defaults for the most popular http servers.

## Authors

---



### Moritz Firsching

⌚ mo271 ⌂ <https://mo271.github.io/>

Moritz Firsching is software engineer at Google Switzerland, where he works on progressive image formats and font compression.



### Luca Versari

⌚ veluca93

Luca Versari is a software engineer at Google, working on JPEG XL<sup>40</sup>. He's finishing a PhD on graph compression and has a background in mathematics.



### Sami Boukortt

⌚ sboukortt

Sami joined Google after completing his studies in engineering mathematics. After a few years of remote interest in compression, he eventually made it his full-time subject of work in 2018.



### Jyrki Alakuijala

⌚ jyrkialakuijala

Jyrki Alakuijala is an active member of the open source software community, and a data compression researcher. Jyrki works at Google as a Technical Lead/Manager, and his recent published work has been with Zopfli, Butteraugli, Guetzli, Gipfeli, WebP lossless, and Brotli 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.

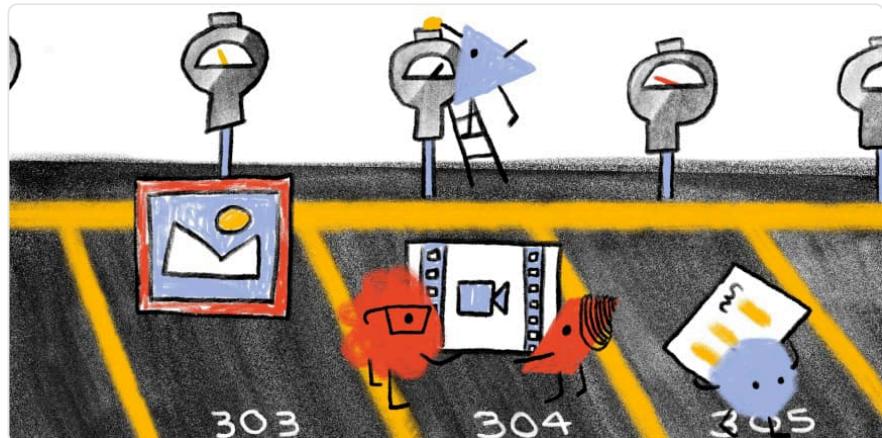
---

40. <https://gitlab.com/wg1/jpeg-xl>



# Part IV Chapter 20

# Caching [UNEDITED]



Written by **Rory Hewitt and Raghu Ramakrishnan**

Reviewed by **Harry Roberts, jzyang, Jai Santhosh, and Soham-S-Sarkar**

Analyzed by **Raghu Ramakrishnan**

## Introduction

Caching is a technique that enables the reuse of previously downloaded content. It involves something (a server which builds web pages, a proxy such as a CDN or the browser itself) storing 'content' (web pages, CSS, JS, images, fonts, etc.) and tagging it appropriately, so it can be reused.

Here's a very high-level example:

Jane visits the home page of the [www.example.com](http://www.example.com) website. Jane lives in Los Angeles, CA, and the `example.com` server is located in Boston, MA. Jane visiting `www.example.com` involves a network request which has to travel across the country.

On the `example.com` server (a.k.a. Origin server), the home page is retrieved. The server knows Jane is located in LA and adds dynamic content to the page - a list of upcoming events near her. Then the page is sent back across the country to Jane and displayed on her browser.

If there is no caching, if Carlos in LA also visits [www.example.com](http://www.example.com) after Jane, his request must travel across the country to the example.com server. The server has to build the same page, including the LA events list. It will have to send the page back to Carlos.

Worse, if Jane revisits the example.com home page, her subsequent requests will act like the first - the request must go across the country and the example.com server must rebuild the home page to send it back to her.

So without any caching, the example.com server builds each request from scratch. That's bad for the server because it's more work. Additionally, any communication between either Jane or Carlos and the example.com server requires data to travel across the country. All of this can add up to a slow experience that's bad for both of them.

However, with server caching, when Jane makes her first request the server builds the LA variant of the home page. It caches the data for reuse by all LA visitors. So when Carlos's request gets to the example.com server, the server checks if it has the LA variant of the home page in its cache. Since that page is in cache as a result of Jane's earlier request, the server saves time by returning the cached page.

More importantly, with browser caching, when Jane's browser receives the page from the server for the first request, it caches the page. All of her future requests for the example.com home page will be served instantly from her browser's cache, without a network request. The example.com server also benefits by not having to process or deal with Jane's request.

Jane is happy. Carlos is happy. The example.com folks are happy. Everyone is happy.

It should be clear then, that browser caching provides a significant performance benefit by avoiding costly network requests. It also helps an application scale by reducing the traffic to a website's origin infrastructure. Server caching also significantly reduces the load on the underlying application.

Caching benefits both the end users (they get their web pages quickly) and the companies serving the web pages (reducing the load on their servers). Caching really is a win-win!

Web architectures typically involve multiple tiers of caching. There are four main places ('caching entities') where caching can occur:

1. An end user's web browser.
2. A service worker cache running in the end user's web browser.
3. A Content Delivery Network (CDN) or similar proxy, which sits between the end user's web browser and the origin server.
4. The origin server itself.

In this chapter, we will primarily be discussing caching within web browsers (1-2), as opposed to caching at the origin server or in a CDN. Nevertheless, many of the specific caching topics discussed in this chapter rely on the relationship between the browser and the server (or CDN, if one is used).

The key to understanding how caching (and the web) works is to remember that it all consists of transactions between a requesting entity (e.g. a browser) and a responding entity (e.g. a server). Each transaction consists of two parts:

1. The request from the requesting entity ("I want object X"), and
2. The response from the responding entity ("Here is object X").

When we talk about caching, it refers to the object (HTML page, image, etc.) cached by the requesting entity.

Below figure shows how a typical request/response flow works for an object (e.g. a web page). A CDN sits between the browser and the server. Note that at each point in the browser → CDN → server flow, each of the caching entities first checks whether it has the object in its cache. It returns the cached object to the requester if found, before forwarding the request to the next caching entity in the chain:

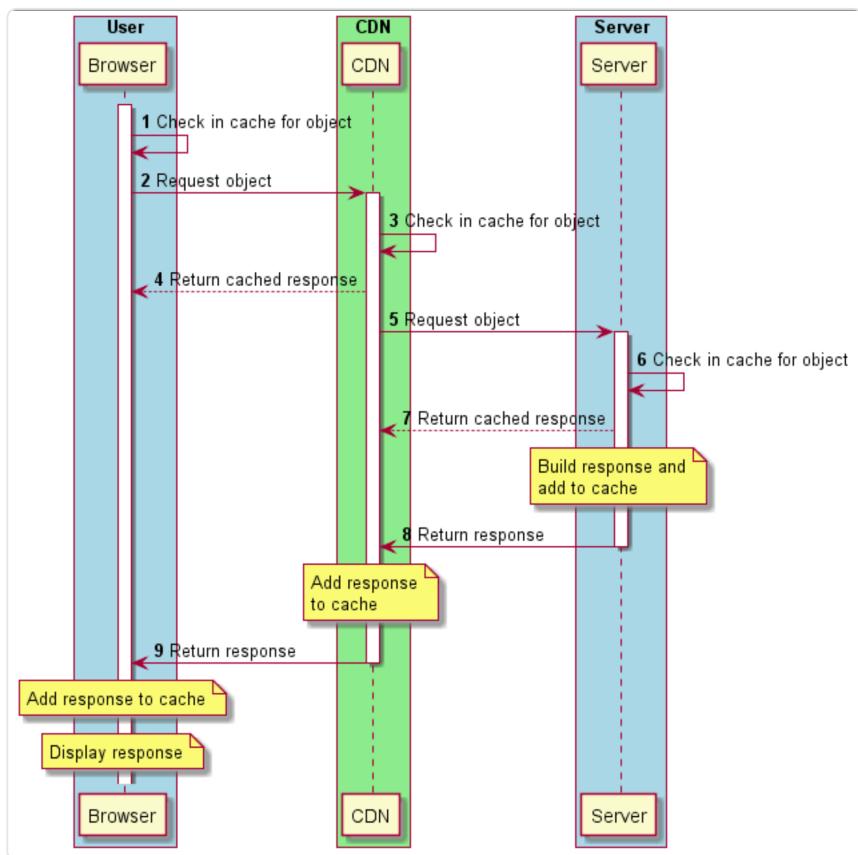


Figure 20.1. Request/response flow for an object.

*Unless specified otherwise, all statistics in this chapter are for desktop, on the understanding that mobile statistics are similar. Where mobile and desktop statistics differ significantly, that is called out. Many of the responses used in this chapter are from web servers which use commonly-available server packages. While we may indicate 'best practices', the practices may not be possible if the software package used has a limited number of cache options.*

## Caching guiding principles

There are three guiding principles to caching web content:

- Cache as much as you can
- Cache for as long as you can
- Cache as close as you can to end users

### Cache as much as you can

When considering what to cache, it is important to understand whether the response content is *static* or *dynamic*.

- An example of static content is an image. For instance, a picture of a cat is the same regardless of who's requesting it or where the requester is located.
- An example of dynamic content is a list of events which are specific to a geographic location. The list will be different based on the requester's location.

Static content is typically cacheable and often for long periods of time. It has a one-to-many relationship between the content (one) and the requests (many).

Dynamically generated content can be more nuanced and requires careful consideration. Some dynamic content can be cached, but often for a shorter period of time. The example of a list of upcoming events will change, possibly from day to day. Different variants of the list may also need to be cached and what's cached in a user's browser may be a subset of what's cached on the server or CDN. Nevertheless, it is possible to cache some dynamic contents. It is incorrect to assume that "dynamic" is another word for "uncacheable".

### Cache for as long as you can

The length of time you would cache a resource is highly dependent on the content's *volatility* (the likelihood and/or frequency of change). For example, an image or a versioned JavaScript file could be cached for a very long time. An API response or a non-versioned JavaScript file may need a shorter cache duration to ensure users get the most up-to-date response. Some content might only be cached for a minute or less. And, of course, some content should not be cached at all. This is discussed in more detail in Identifying caching opportunities.

Another point to bear in mind is that no matter how long you tell a browser to cache content for, the browser may evict that content from cache before that point in time. It may do so to make room for other content that is accessed more frequently, etc.. However, a browser will never cache content for longer than it is told.

## **Cache as close to end users as you can**

Caching content close to the end user reduces download times by removing latency. For example, if a resource is cached in a user's browser, then the request never goes out to the network and it is available instantaneously every time the user needs it. For visitors that don't have entries in their browser's cache, a CDN would be the next place a cached resource is returned from. In most cases, it will be faster to fetch a resource from a local cache or a CDN compared to an origin server.

## **Some terminology**

- Caching entity - the hardware or software that is doing the caching. Due to the focus of this chapter, we use "browser" as a synonym for "caching entity" unless otherwise specified.
- TTL - the Time-To-Live of a cached object defines how long it can be stored in a cache, typically measured in seconds. After a cached object reaches its TTL, it is marked as 'stale' by the cache. Depending on how it was added to the cache (see the details of the caching headers below), it may be evicted from cache immediately, or it may remain in the cache but marked as a 'stale' object, requiring revalidation before reuse.
- Eviction - the automated process by which an object is actually removed from a cache when/after it reaches its TTL or possibly when the cache is full.
- Revalidation - a cached object that is marked as stale may need to be 'revalidated' with the server before it can be displayed to the user. The browser must first check with the server that the object the browser has in its cache is still up-to-date and valid.

## **Overview of browser caching**

When a browser makes a request for a piece of content (e.g. a web page), it will receive a response which includes not just the content itself (the HTML markup), but also a number of response headers which describe the content, including information about its cacheability.

The caching-related headers, or the absence of them, tell the browser three important pieces of information:

- **Cacheability:** Is this content cacheable?
- **Freshness:** If it is cacheable, how long can it be cached for?
- **Validation:** If it is cacheable, how do I subsequently ensure that my cached version is still fresh?

The full specifications for these caching headers are in RFC 7234, and discussed in sections 4.2 (Freshness) and 4.3 (Validation).

The two HTTP response headers typically used for specifying freshness are `Cache-Control` and `Expires`:

- `Expires` specifies an explicit expiration date and time (i.e. when exactly the content expires)
- `Cache-Control` specifies a cache duration (i.e. how long the content can be cached in the browser relative to when it was requested)

Often, both these headers are specified; in that case `Cache-Control` takes precedence. These headers are discussed in more detail below.

## **Cache-Control VS Expires**

In the early HTTP/1.0 days of the web, the `Expires` header was the only cache-related response header. As stated above, it is used to indicate the exact date/time after which the response is considered stale. Its value is a date and time, such as:

```
Expires: Thu, 01 Dec 1994 16:00:00 GMT
```

The `Expires` header can be thought of as a 'blunt instrument'. If a relative cache TTL is required, then processing must be done on the server to generate an appropriate value based upon the current date/time.

HTTP/1.1 introduced the `Cache-Control` header, which is supported by all modern browsers. The `Cache-Control` header provides much more extensibility and flexibility than `Expires` via *caching directives*, several of which can be specified together. Details on the various directives are below.

The simple example below shows a request and response for a JavaScript file (some headers have been removed for clarity). The `Date` header indicates the current date (specifically, the date that the content was served). The `Expires` header indicates that it can be cached for 10 minutes (the difference between the `Expires` and `Date` headers). The `Cache-Control` header

specifies the `max-age` directive, which indicates that the resource can be cached for 600 seconds (5 minutes). Since `Cache-Control` takes precedence over `Expires`, the browser will cache the response for 5 minutes, after which it will be marked as stale:

```
> GET /static/js/main.js HTTP/2
> Host: www.example.org
> Accept: */*
< HTTP/2 200
< Date: Thu, 23 Jul 2020 03:04:17 GMT
< Expires: Thu, 23 Jul 2020 03:14:17 GMT
< Cache-Control: public, max-age=600
```

RFC 7234 says that if no caching headers are present in a response, then the browser is allowed to *heuristically* cache the response - it suggests a cache duration of 10% of the time since the `Last-Modified` header (if passed). In such cases, most browsers implement a variation of this suggestion, but some may cache the response indefinitely and some may not cache it at all. Because of this variation between browsers, it is important to explicitly set specific caching rules to ensure that you are in control of the cacheability of your content.

- 73.6% of responses are served with a `Cache-Control` header
- 55.5% of responses are served with an `Expires` header
- 54.8% of responses include both headers
- 25.6% of responses did not include either header, and are therefore subject to heuristic caching

## Usage of Cache-Control and Expires headers

Web Almanac 2020: Caching

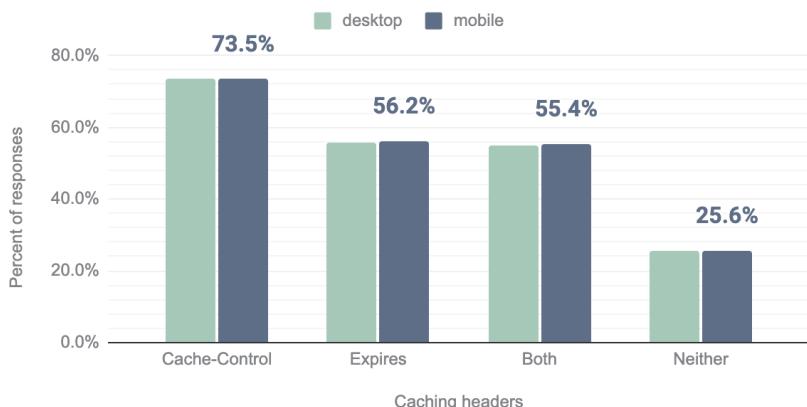


Figure 20.2. Usage of Cache-Control and Expires headers.

## Usage of Cache-Control and Expires headers (2019)

Web Almanac 2020: Caching

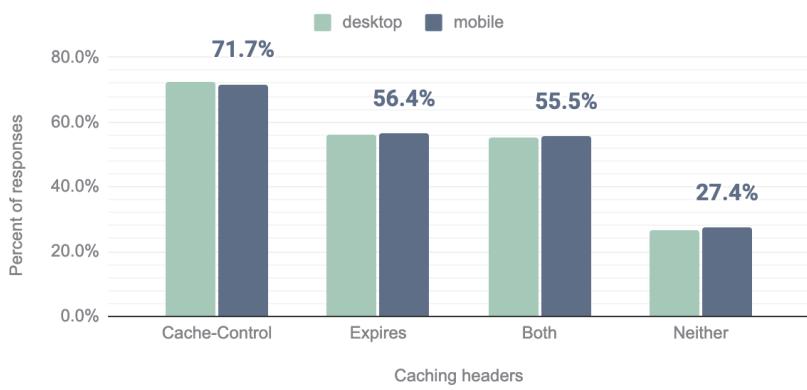


Figure 20.3. Usage of Cache-Control and Expires headers in 2019.

These statistics are interesting since, compared with 2019, while we see an increase in the use of the Cache-Control header (1.3%), we also see a minimal decrease in the use of the older Expires header (0.7%). Effectively, a percentage of servers are merely adding the Cache-Control header to their responses without removing the Expires header.

As we delve into the various directives allowed in the `Cache-Control` header, we will see how its flexibility and power make it a better fit in many cases.

## Cache-Control directives

When you use the `Cache-Control` header, you specify one or more *directives* - predefined values that indicate specific caching functionality. Multiple directives are separated by commas and can be specified in any order, although some of them 'clash' with one another (e.g. `public` and `private`). Some directives take a value, such as `max-age`.

Below is a table showing the most common `Cache-Control` directives:

Directive	Description
<code>max-age</code>	Indicates the number of seconds that a resource can be cached for, relative to the current time. For example ' <code>max-age=86400</code> '.
<code>public</code>	Any cache may store the response, including the browser, and any proxies between the server and the browser, such as a CDN. This is assumed by default.
<code>no-cache</code>	A cached entry must be revalidated prior to its use, via a conditional request, even if it is not marked as stale.
<code>must-revalidate</code>	A stale cached entry must be revalidated prior to its use, via a conditional request.
<code>no-store</code>	Indicates that the response must not be cached.
<code>private</code>	The response is intended for a specific user and should not be stored by shared caches such as proxies and CDNs.
<code>proxy-revalidate</code>	Same as <code>must-revalidate</code> but applies to shared caches.
<code>s-maxage</code>	Same as <code>max-age</code> but applies to shared caches (e.g. CDN's) only.
<code>immutable</code>	Indicates that the cached entry will never change during its TTL, and that revalidation is not necessary.
<code>stale-while-revalidate</code>	Indicates that the client is willing to accept a stale response while asynchronously checking in the background for a fresh one.
<code>stale-if-error</code>	Indicates that the client is willing to accept a stale response if the check for a fresh one fails.

Figure 20.4. `Cache-Control` directives.

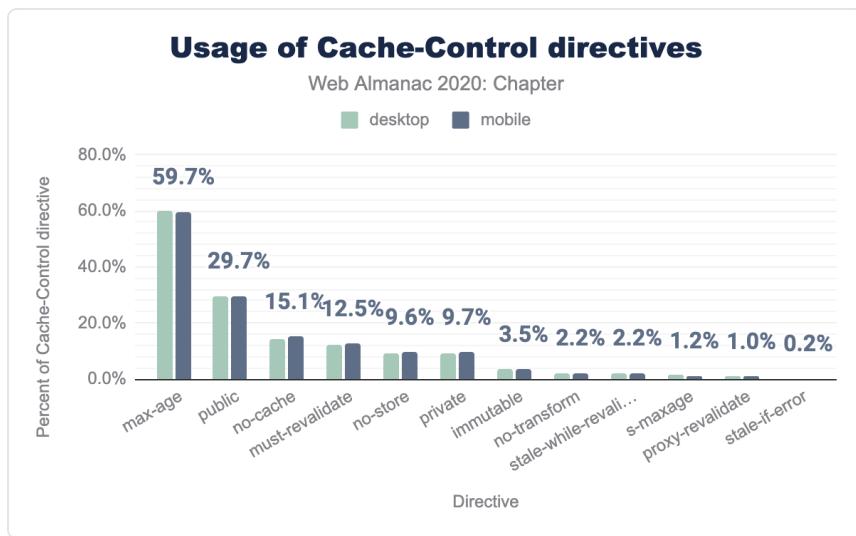
The `max-age` directive is the most commonly-found, since it directly defines the TTL, in the same way that the `Expires` header does.

Here is an example of a valid Cache-Control header with multiple directives:

```
Cache-Control: public, max-age=86400, must-revalidate
```

This indicates that the object can be cached for 86,400 seconds (1 day) and it can be stored by all caches between the server and the browser, as well as in the browser itself. Once it has reached its TTL and is marked as stale, it can remain in cache, but must be conditionally revalidated before reuse.

- 60.2% of responses include a `Cache-Control` header with the `max-age` directive.
- 45.5% of responses include the `Cache-Control` header with the `max-age` directive and the `Expires` header, which means that 10% of responses are caching solely based on the older `Expires` header.



*Figure 20.5. Distribution of `Cache-Control` directives.*

The above figure illustrates the 11 `Cache-Control` directives in use on mobile and desktop websites. There are a few interesting observations about the popularity of these cache directives:

- `max-age` is used by about 60.2% of `Cache-Control` headers, and `no-store` is used by about 9.2% (see below for some discussion on the meaning and use of the `no-store` directive).
- Explicitly specifying `public` isn't ever really necessary since cached entries are assumed `public` unless `private` is specified. Nevertheless, almost one third of responses include `public` - a waste of a few header bytes on every response :)

- The `immutable` directive is relatively new, introduced in 2017 and is only supported on Firefox and Safari - its usage is still only at about 3.5%, but it is widely seen in responses from Facebook, Google, Wix, Shopify and others. It has the potential to greatly improve cacheability for certain types of requests.

As we head out to the long tail, there are a small percentage of 'invalid' directives that can be found; these are ignored by browsers, and just end up wasting header bytes. Broadly they fall into two categories:

- Misspelled directives such as `nocache` and `s-max-age` and invalid directive syntax, such as using `:` instead of `=` or using `_` instead of `-`.
- Non-existent directives such as `max-stale`, `proxy-public`, `surrogate-control`.

The most interesting standout in the list of invalid directives is the use of `no-cache="set-cookie"` (even at only 0.2% of all `Cache-Control` header values, it still makes up more than all the other invalid directives combined). In some early discussions on the `Cache-Control` header, this syntax was raised as a possible way to ensure that any `Set-Cookie` response headers (which might be user-specific) would not be cached with the object itself by any intermediate proxies such as CDNs. However, this syntax was not included in the final RFC; nearly equivalent functionality can be implemented using the `private` directive, and the `no-cache` directive does not allow a value.

## **Cache-Control: no-store, no-cache and max-age=0**

When a response absolutely must not be cached, the `Cache-Control no-store` directive should be used; if this directive is not specified, then the response *is considered cacheable and may be cached*. Note that if `no-store` is specified, it takes precedence over other directive - this makes sense, since serious privacy and security issues could occur if a resource is cached which should not be.

We can see a few common errors that are made when attempting to configure a response to be non-cacheable:

- Specifying `Cache-Control: no-cache` may sound like a directive to not cache the resource. However, as noted above, the `no-cache` directive does allow the resource to be cached - it simply informs the browser to revalidate the resource prior to use and is not the same as stopping the resource from being cached at all.
- Setting `Cache-Control: max-age=0` sets the TTL to 0 seconds, but again, that is not the same as being non-cacheable. When `max-age=0` is specified, the resource is cached, but is marked as stale, resulting in the browser having to immediately revalidate its freshness.

Functionally, `no-cache` and `max-age=0` are similar, since they both require revalidation of a cached resource. The `no-cache` directive can also be used alongside a `max-age` directive that is greater than 0 - this results in the object being cached for the specified TTL, but being revalidated prior to every use.

When looking at the above three discussed directives, 2.3% of responses include the combination of all three `no-store`, `no-cache` and `max-age=0` directives, 6.6% of responses include both `no-store` and `no-cache`, and a negligible number of responses (< 1%) include `no-store` alone.

As noted above, where `no-store` is specified with either/both of `no-cache` and `max-age=0`, the `no-store` directive takes precedence, and the other directives are ignored. Therefore, if you don't want content to be cached anywhere, simply specifying `Cache-Control: no-store` is sufficient, and is both simple and uses the minimum number of header bytes.

The `max-age=0` directive is present on less than 2% of responses where `no-store` is not specified. In such cases, the resource will be cached in the browser but will require revalidation as it is immediately marked as stale.

## Conditional requests and revalidation

There are often cases where a browser has previously requested an object and already has it in its cache but the cache entry has already exceeded its TTL (and is therefore marked as stale) or where the object is defined as one that must be revalidated prior to use.

In these cases, the browser can make a conditional request to the server - effectively saying "*I have object X in my cache - can I use it, or do you have a more recent version I should use instead?*". The server can respond in one of two ways:

- "*Yes, the version of object X you have in cache is fine to use*" - in this case the server response consists of a `304 Not Modified` status code and response headers, but no response body
- "*No, here is a more recent version of object X - use this instead*" - in this case the server response consists of a `200 OK` status code, response headers, and a new response body (the actual new version of object X)

In either case, the server can optionally include updated caching response headers, possibly extending the TTL of the object so the browser can use the object for a further period of time without needing to make more conditional requests.

The above is known as *revalidation* and if implemented correctly can significantly improve perceived performance - since a `304 Not Modified` response consists only of headers, it is much smaller than a `200 OK` response, resulting in reduced bandwidth and a quicker response.

So how does the server identify a conditional request from a regular request?

It actually all comes down to the initial request for the object. When a browser requests an object which it does not already have in its cache, it simply makes a GET request, like this (some headers removed for clarity):

```
> GET /index.html HTTP/2
> Host: www.example.org
> Accept: */*
```

If the server wants to allow the browser to make use of conditional requests (this decision is entirely up to the server!), it can include one or both of two response headers which identify the object as being eligible for subsequent conditional requests. The two response headers are:

- `Last-Modified` - this indicates when the object was last changed. Its value is a date timestamp.
- `ETag (Entity Tag)` - this provides a unique identifier for the content as a quoted string. It can take any format that the server chooses; it is typically a hash of the file contents, but it could be a timestamp or a simple string.

If both headers are present, `ETag` takes precedence.

### **Last-Modified**

When the server receives the request for the file, it can include the date/time that the file was most recently changed as a response header, like this:

```
< HTTP/2 200
< Date: Thu, 23 Jul 2020 03:04:17 GMT
< Last-Modified: Mon, 20 Jul 2020 11:43:22 GMT
< Cache-Control: max-age=600

< ...lots of html here...
```

The browser will cache this object for 600 seconds (as defined in the `Cache-Control` header), after which it will mark the object as stale. If the browser needs to use the file again, it requests the file from the server just as it did initially, but this time it includes an additional request header, called `If-Modified-Since`, which it sets to the value that was passed in the `Last-`

Modified response header in the initial response:

The browser will cache this object for 600 seconds (as defined in the Cache-Control header), after which it will mark the object as stale. If the browser needs to use the file again, it requests the file from the server just as it did initially, but this time it includes an additional request header, called If-Modified-Since, which it sets to the value that was passed in the Last-Modified response header in the initial response:

```
> GET /index.html HTTP/2
> Host: www.example.org
> Accept: /*
> If-Modified-Since: Mon, 20 Jul 2020 11:43:22 GMT
```

When the server receives this request, it can check whether the object has changed by comparing the If-Modified-Since header value with the date that it most recently changed the file.

If the two values are the same, then the server knows that the browser has the latest version of the file and the server can return a 304 Not Modified response with just headers (including the same Last-Modified header value) and no response body:

```
< HTTP/2 304
< Date: Thu, 23 Jul 2020 03:14:17 GMT
< Last-Modified: Mon, 20 Jul 2020 11:43:22 GMT
< Cache-Control: max-age=600
```

However, if the file on the server has changed since it was last requested by the browser, then the server returns a 200 OK response consisting of headers (including an updated Last-Modified header) and the new version of the file in the body:

```
< HTTP/2 200
< Date: Thu, 23 Jul 2020 03:14:17 GMT
< Last-Modified: Thu, 23 Jul 2020 03:12:42 GMT
< Cache-Control: max-age=600

< ...lots of html here...
```

As you can see, the `Last-Modified` response header and `If-Modified-Since` request header work as a pair.

## ETag

The functionality here is almost exactly the same as the date-based `Last-Modified`/`If-Modified-Since` conditional request processing described above.

However, in this case, the Server sends an `ETag` response header - rather than a date timestamp, an `ETag` is simply a string - often a hash of the file contents or a version number calculated by the server. The format of this string is entirely up to the server - the only important fact is that the server changes the `ETag` value whenever it changes the file.

In this example, when the server receives the initial request for the file, it can return the file's version in an `ETag` response header, like this:

```
< HTTP/2 200
< Date: Thu, 23 Jul 2020 03:04:17 GMT
< ETag: "v123.4.01"
< Cache-Control: max-age=600

< ...lots of html here...
```

As with the `If-Modified-Since` example above, the browser will cache this object for 600 seconds, as defined in the `Cache-Control` header. When it needs to request the object from the server again, it includes an additional request header, called `If-None-Match`, which has the value passed in the `ETag` response header in the initial response:

```
> GET /index.html HTTP/2
> Host: www.example.org
> Accept: */
> If-None-Match: "v123.4.01"
```

When the server receives this request, it can check whether the object has changed by comparing the `If-None-Match` header value with the current version it has of the file. If the two values are the same, then the browser has the latest version of the file and the server can return a `304 Not Modified` response with just headers:

```
< HTTP/2 304
< Date: Thu, 23 Jul 2020 03:14:17 GMT
< ETag: "v123.4.01"
< Cache-Control: max-age=600
```

However, if the values are different, then the version of the file on the server is more recent than the version that the browser has, so the server returns a 200 OK response consisting of headers (including an updated ETag header) and the new version of the file:

```
< HTTP/2 200
< Date: Thu, 23 Jul 2020 03:14:17 GMT
< ETag: "v123.5.06"
< Cache-Control: public, max-age=600

< ...lots of html here...
```

Again, we see a pair of headers being used for this conditional request processing - the ETag response header and the If-None-Match request header.

In the same way that the Cache-Control header has more power and flexibility than the Expires header, the ETag header is in many ways an improvement over the Last-Modified header. There are two reasons for this:

1. The server can define its own format for the ETag header. The example above shows a version string, but it could be a hash, or a random string. By allowing this, versions of an object are not explicitly linked to dates, and this allows a server to create a new version of a file and yet give it the same ETag as the prior version - perhaps if the file change is unimportant
2. ETags can be defined as either 'strong' or 'weak', which allows browsers to validate them differently. A full understanding and discussion of this functionality is beyond the scope of this chapter, but can be found in RFC 7232.
  - 73.5% of responses are served with a Last-Modified header. Its usage has marginally increased (by < 1%) in comparison to 2019.
  - 47.9% of responses are served with an ETag header. Out of these responses, 36% are 'strong', 98.2% are 'weak', and the remaining 1.8% are invalid. In contrast with Last-Modified, the usage of ETag headers has marginally decreased (by <1%) in comparison to 2019.
  - 42.8% of responses are served with both headers (as noted above, in this case, the

ETag header takes precedence).

- 21.4% of responses include neither a Last-Modified or ETag header.

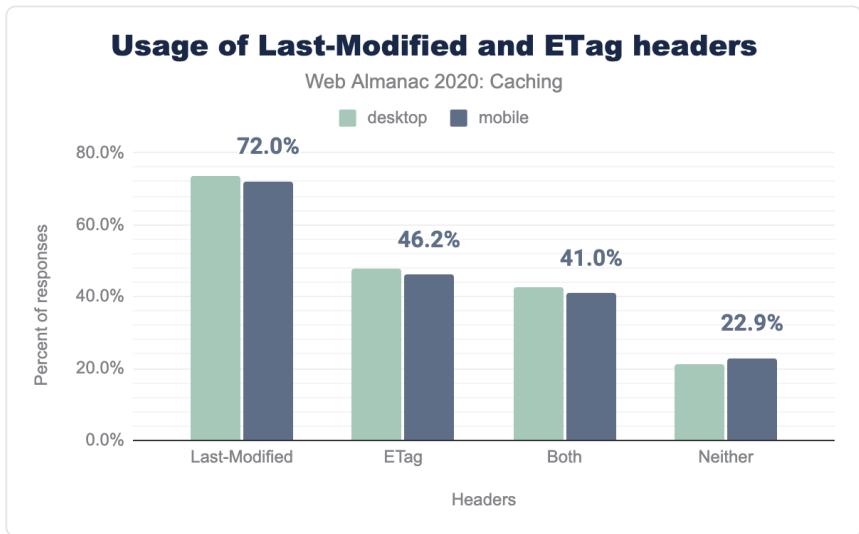


Figure 20.6. Adoption of validating freshness via `Last-Modified` and `ETag` headers.

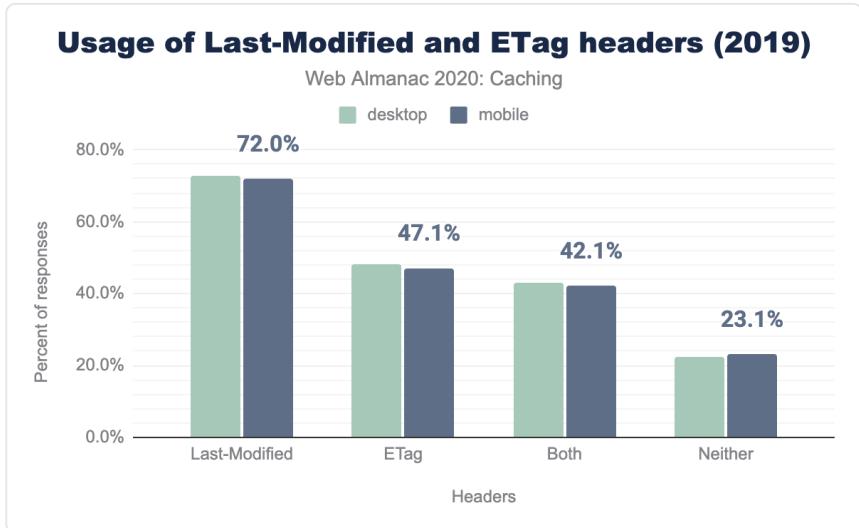


Figure 20.7. Adoption of validating freshness via `Last-Modified` and `ETag` headers in 2019.

Correctly-implemented revalidation using conditional requests can significantly reduce bandwidth (304 responses are typically much smaller than 200 responses), load on servers

(only a small amount of processing is required to compare change dates or hashes) and improve perceived performance (servers respond more quickly with a 304). However, as we can see from the above statistics, more than a fifth of all requests are not using any form of conditional requests.

- Only 0.1% of the responses had a 304 Not Modified status.
- 20.5% of the responses had no ETag header and contained the same Last-Modified value, passed in the If-Modified-Since header of the corresponding request. Out of these, 86% had a 304 Not Modified status.
- 86.1% of the responses contained the same ETag value, passed in the If-None-Match header of the corresponding request. If the If-Modified-Since header is also present, ETag takes precedence. Out of these, 88.9% had a 304 Not Modified status.

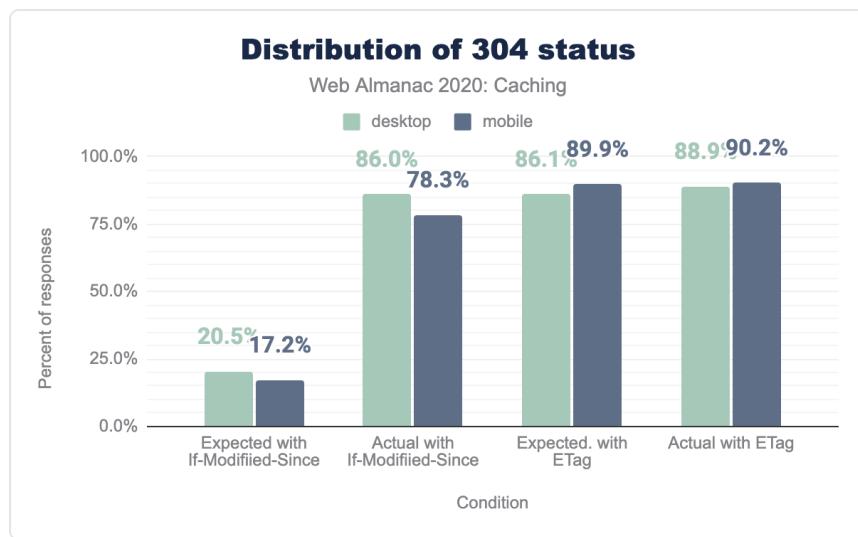


Figure 20.8. Distribution of 304 Not Modified status.

## Validity of date strings

Throughout this document, we have discussed several caching-related HTTP headers used to convey timestamps:

- The Date response header indicates when the resource was served to a client.
- The Last-Modified response header indicates when a resource was last changed on the server.

- The `Expires` header is used to indicate for how long a resource is cacheable.

All three of these HTTP headers use a date formatted string to represent timestamps. The date-formatted string is defined in RFC 2616, and must specify the 'GMT' timestamp string. For example:

```
> GET /index.html HTTP/2
> Host: www.example.org
> Accept: /*

< HTTP/2 200
< Date: Thu, 23 Jul 2020 03:14:17 GMT
< Cache-Control: max-age=600
< Last-Modified: Mon, 20 Jul 2020 11:43:22 GMT
```

Invalid date strings are ignored by most browsers, which can affect the cacheability of the response on which they are served - for example, an invalid `Last-Modified` header will result in the browser being unable to subsequently perform a conditional request for the object, since it is cached without that invalid timestamp.

Because the `Date` HTTP response header is almost always generated automatically by the web server, invalid values are extremely rare. Similarly `Last-Modified` headers had a very low percentage (0.5%) of invalid values. What was very surprising to see though, was that a relatively high 2.9% of `Expires` headers used an invalid date format (2.5% in mobile).

## Invalid date formats in response headers

Web Almanac 2020: Caching

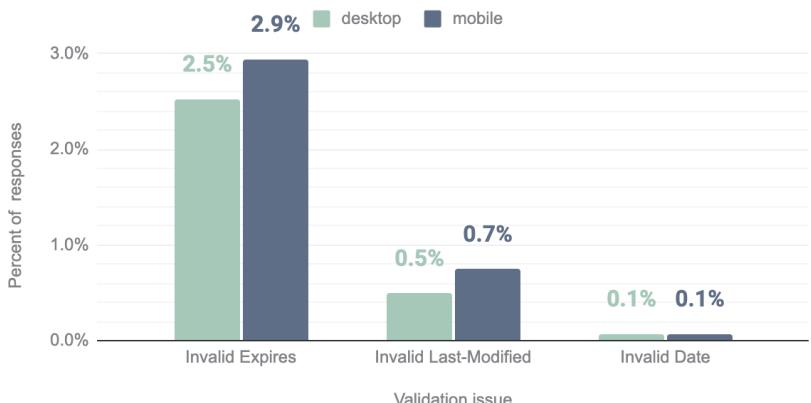


Figure 20.9. Invalid date formats in response headers.

Examples of some of the invalid uses of the `Expires` header are:

- Valid date formats, but using a time zone other than 'GMT'
- Numerical values such as 0 or -1
- Values that would be valid in a `Cache-Control` header

One large source of invalid `Expires` headers is from assets served from a popular third party, in which a date/time uses the EST time zone, for example `Expires: Tue, 27 Apr 1971 19:44:06 EST`. Note that some browsers may understand and accept this date format, on the principle of robustness, but it should not be assumed that this will be the case.

## The `vary` header

We have discussed how a caching entity can determine whether a response object is cacheable, and for how long it can be cached. However, one of the most important steps the caching entity must take is determining if the resource being requested is already in its cache. While this may seem simple, many times the URL alone is not enough to determine this. For example, requests with the same URL could vary in what compression they used (gzip, brotli, etc.) or could be returned in different encodings (XML, JSON etc.).

To solve this problem, when a caching entity caches an object, it gives the object a unique identifier (a cache key). When it needs to determine whether the object is in its cache, it checks for the existence of the object using the cache key as a lookup. By default, this cache key is

simply the URL used to retrieve the object, but servers can tell the caching entity to include other 'attributes' of the response (such as compression method) in the cache key, by including the `Vary` response header, to ensure that the correct object is subsequently retrieved from cache - the `Vary` header identifies 'variants' of the object, based on factors other than the URL.

The `Vary` response header instructs the browser to add the value of one or more request header values to the cache key. The most common example of this is `Vary: Accept-Encoding`, which will result in the browser caching the same object in different formats, based on the different `Accept-Encoding` request header values (i.e. `gzip`, `br`, `deflate`).

A caching entity sends a request for an HTML file, indicating that it will accept a zipped response:

```
> GET /index.html HTTP/2
> Host: www.example.org
> Accept-Encoding: gzip
```

The server responds with the object, and indicates that the version it is sending should include the value of the `Accept-Encoding` request header.

```
< HTTP/2 200 OK
< Content-Type: text/html
< Vary: Accept-Encoding
```

In this simplified example, the caching entity would cache the object using a combination of the URL and the `Vary` header.

Another common value is `Vary: Accept-Encoding, User-Agent`, which instructs the client to include both the `Accept-Encoding` and `User-Agent` values in the cache key. When used from a browser, this might not make much sense - each browser has its own `User-Agent` value, so a browser would not make a request using different `User-Agent` values anyway. However, when discussing shared proxies and CDNs, using values other than `Accept-Encoding` can be problematic as it dilutes ('fragments') the cache and can reduce the amount of traffic served from cache. For instance, if a CDN attempts to cache many different variants of an object, including not just the URL and the `Accept-Encoding` header but also the `User-Agent` string (of which there are several thousand different varieties), it may end up filling up the cache with many almost identical (or indeed, identical) cached objects. This is very inefficient, and can lead to very sub-optimal caching within the CDN, resulting in fewer cache hits and greater latency. In general, you should only vary the cache if you are serving alternate content to clients based on that header.

The `Vary` header is used on 43.4% of HTTP responses, and 84.2% of these responses include a `Cache-Control` header.

The graph below details the popularity for the top 10 `Vary` header values. `Accept-Encoding` accounts for almost 92% of `Vary`'s use, with `User-Agent` at 10.7%, `Origin` (used for CORS processing) at 8%, and `Accept` at 4.1% making up much of the rest.

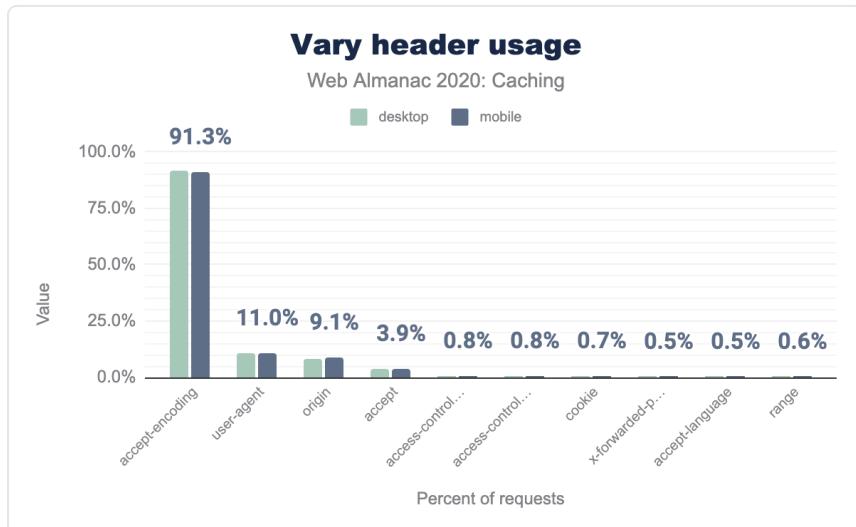


Figure 20.10. `Vary header usage`.

## Setting cookies on cacheable responses

When a response is cached, its entire set of response headers are included with the cached object as well. This is why you can see the response headers when inspecting a cached response in Chrome via DevTools:

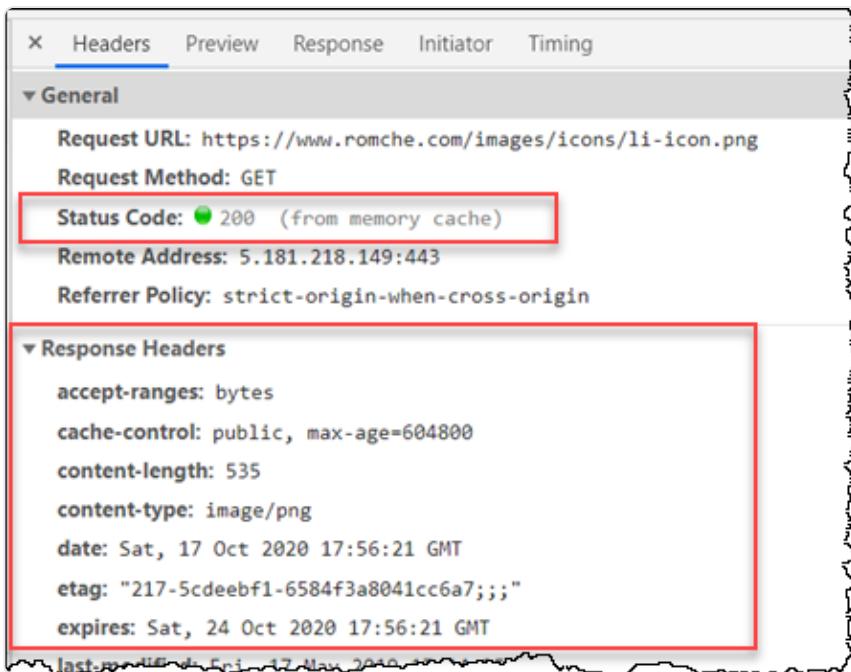


Figure 20.11. Chrome Dev Tools for a cached resource.

But what happens if you have a `Set-Cookie` on a response? According to RFC 7234 Section 8, the presence of a `Set-Cookie` response header does not inhibit caching. This means that a cached entry might contain a `Set-Cookie` response header. The RFC goes on to recommend that you should configure appropriate `Cache-Control` headers to control how responses are cached.

Since we have primarily been talking about browser caching, you may think this isn't a big issue - the `Set-Cookie` response headers that were sent by the server to me in responses to my requests clearly contain my cookies, so there's no problem if my browser caches them. However, if there is a CDN between myself and the server, the server must indicate to the CDN that the response should not be cached in the CDN itself, so that the response meant for me is not cached and then served (including my `Set-Cookie` headers!) to other users.

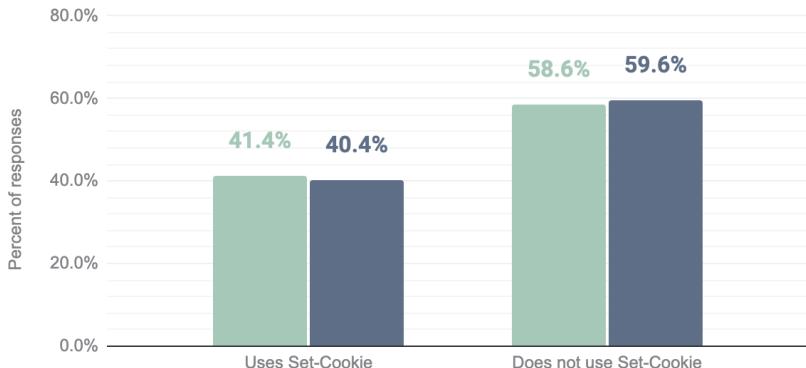
For example, if a login cookie or a session cookie is present in a CDN's cached object, then that cookie could potentially be reused by another client. The primary way to avoid this is for the server to send the `Cache-Control: private` directive, which tells the CDN not to cache the response, because it may only be cached by the client browser.

41.4% of cacheable responses contain a `Set-Cookie` header. Of those responses, only 4.6% use the `private` directive. The remaining 95.4% (189.2 million HTTP responses) contain at

least one `Set-Cookie` response header and can be cached by both public cache servers, such as CDNs. This is concerning and may indicate a continued lack of understanding about how cacheability and cookies coexist.

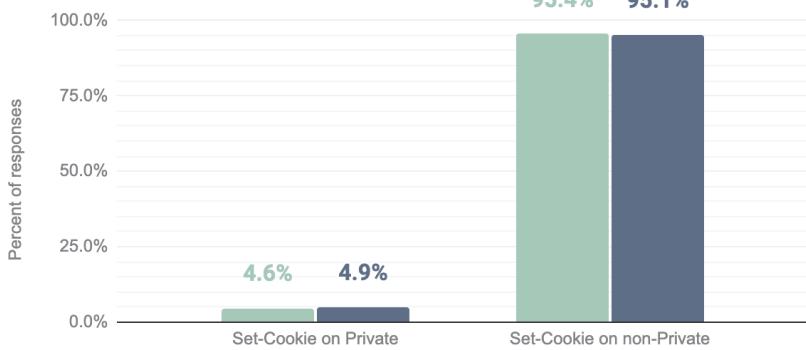
### Set-Cookie usage on cacheable responses

Web Almanac 2020: Caching


Figure 20.12. `Set-Cookie` in cacheable responses.

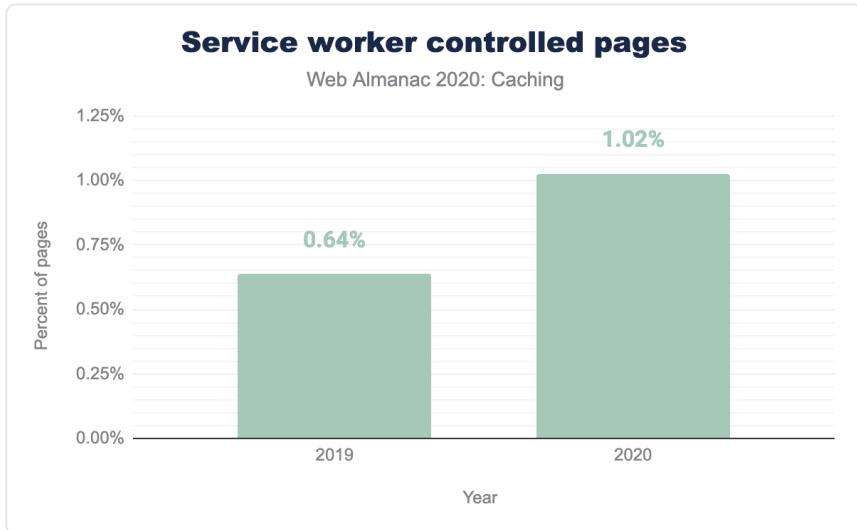
### Set-Cookie usage on private and non-private cacheable responses

Web Almanac 2020: Caching


Figure 20.13. `Set-Cookie` in private and non private cacheable responses.

## Service workers

Service workers are a feature of HTML5 that allow front-end developers to specify scripts that should run outside the 'normal' request/response flow of web pages, communicating with the web page via messages. Common uses of service workers are for background synchronization and push notifications and, obviously, for caching - and browser support has been rapidly growing for them.



*Figure 20.14. Growth in service worker controlled pages from 2019.*

Adoption is just at 1% of websites, but it has been steadily increasing since July 2019. The Progressive Web App chapter discusses this more, including the fact that it is used a lot more than this graph suggests due to its usage on popular sites, which are only counted once in the above graph.

In the table below, you can see that out of a total of 6,225,774 websites, only 64,373 (1%) have implemented a service worker.

Sites not using service workers	Sites using service workers	Total sites
6,225,774	64,373	6,290,147

*Figure 20.15. Number of websites using service workers.*

If we break this out by HTTP vs HTTPS, then this gets even more interesting. Even though HTTPS is a requirement for using service workers, the following table shows that 1,469 of the

sites using them are served over HTTP.

HTTP Sites	HTTPS Sites	Total Sites
1,469	62,904	64,373

Figure 20.16. Number of websites using service workers by HTTP/HTTPS.

## What type of content are we caching?

As we have seen, a cacheable resource is stored by the browser for a period of time and is available for reuse on subsequent requests. Across all HTTP(S) requests, 90.8% of responses are considered cacheable, meaning that a cache is permitted to store them. Out of these,

- 4.2% of requests have a TTL of 0 seconds, which causes the object to be added to cache, but immediately marked as stale, requiring revalidation.
- 28.2% are cached heuristically because of a lack of either a `Cache-Control` or `Expires` header.
- 59.4% are cached for more than 0 seconds.

The remaining 9.2% of responses are not permitted to be stored in browser caches - typically because of `Cache-Control: no-store`.

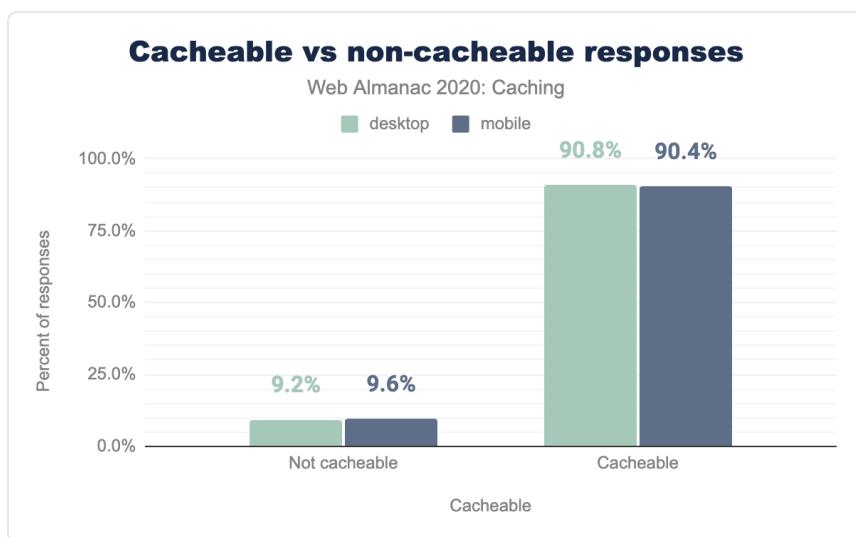


Figure 20.17. Distribution of cacheable and non-cacheable responses.

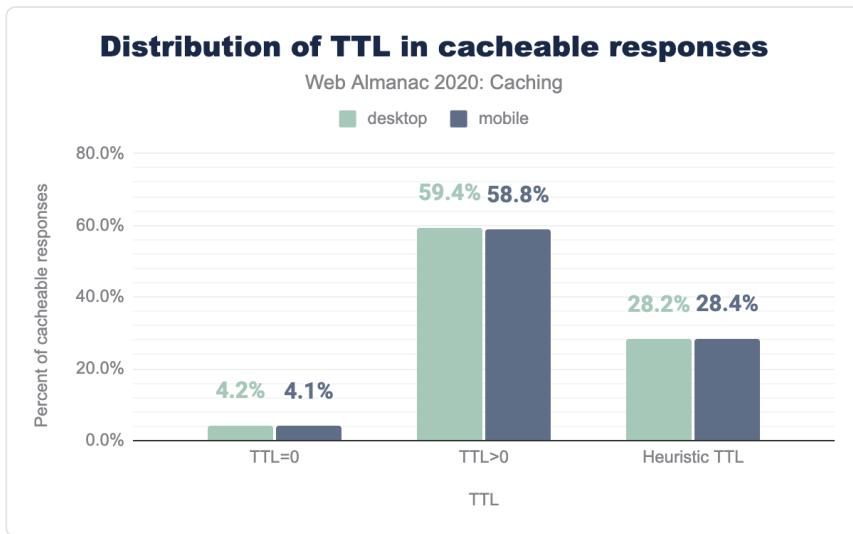


Figure 20.18. Distribution of TTL in cacheable responses.

The table below details the cache TTL values for desktop requests by type. Most content types are being cached, however CSS resources are consistently cached with high TTLs.

	Cache TTL percentiles (in hours)				
	10	25	50	75	90
audio	6	6	240	720	8,760
css	24	24	720	8,760	8,760
font	720	8,760	8,760	8,760	8,760
html	0	1	336	8,760	87,600
image	4	168	720	8,760	8,766
other	0	1	30	240	8,760
script	0	2	720	8,760	8,760
text	0	1	6	6	720
video	6	12	336	336	8,760
xml	0	24	24	24	8,760

Figure 20.19. Desktop cache TTL percentiles by resource type.

While most of the median TTLs are high, the lower percentiles highlight some of the missed caching opportunities. For example, the median TTL for images is 720 hours (1 month); however the 25<sup>th</sup> percentile is just 168 hours (1 week) and the 10<sup>th</sup> percentile has dropped to just a few hours. Compare this with fonts, which have a very high TTL of 8760 hours (1 year) all the way down to the 25th percentile, with even the 10<sup>th</sup> percentile showing a TTL of 1 month.

By exploring the cacheability by content type in more detail in figure below, we can see that while fonts, video and audio, and CSS files are browser cached at close to 100% (which makes sense, since these files are typically very static), approximately one third of all HTML responses are considered non-cacheable.

Additionally, 13.6% of images and scripts are non-cacheable. There is likely some room for improvement here, since no doubt some of these objects are also static and could be cached at a higher rate - remember: *cache as much as you can for as long as you can!*

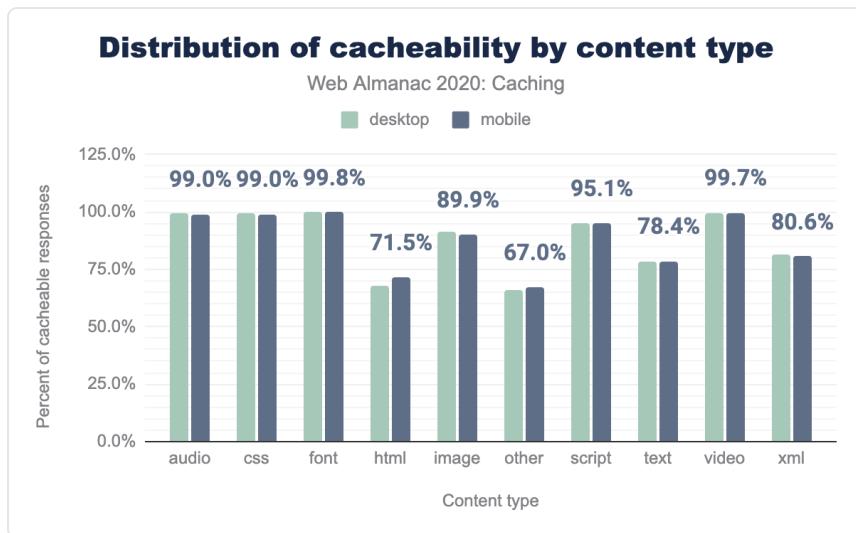


Figure 20.20. Distribution of cacheability by content type.

## How do cache TTLs compare to resource age?

So far we've talked about how servers tell a client what is cacheable, and how long it has been cached for. When designing cache rules, it is also important to understand how old the content you are serving is.

When you (the server) are selecting a cache TTL to specify in response headers to send back to the client, ask yourself: "how often am I updating these assets?" and "what is their content

sensitivity?". For example, if a hero image is going to be modified infrequently, then it could be cached with a very long TTL. By contrast, if a JavaScript file will change frequently, then either it should be versioned (for instance, by using an ETag or with a unique query string) and cached with a long TTL or it should be cached with a much shorter TTL.

The graphs below illustrate the relative age of resources by content type. Some of the interesting observations in this data are:

- First party HTML is the content type with the shortest age, with 42.5% of the requests having an age less than a week. In most of the other content types, third party content has a smaller resource age than first party content.
- Some of the longest aged first party content on the web, with age eight weeks or more, are the traditionally cacheable objects like images (78.3%), scripts (68.6%), CSS (74.1%), web fonts (79.3%), audio (77.9%) and video (78.6%).
- There is a significant gap in some first vs. third party resources having an age of more than a week. 93.5% of first party CSS are older than one week compared to 51.5% of 3rd party CSS, which are older than one week.

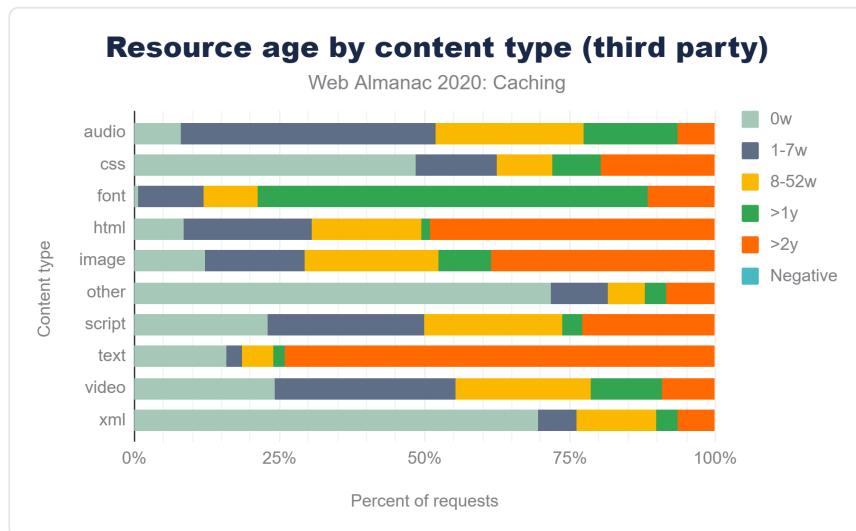


Figure 20.21. Resource age by Content Type (1st Party).

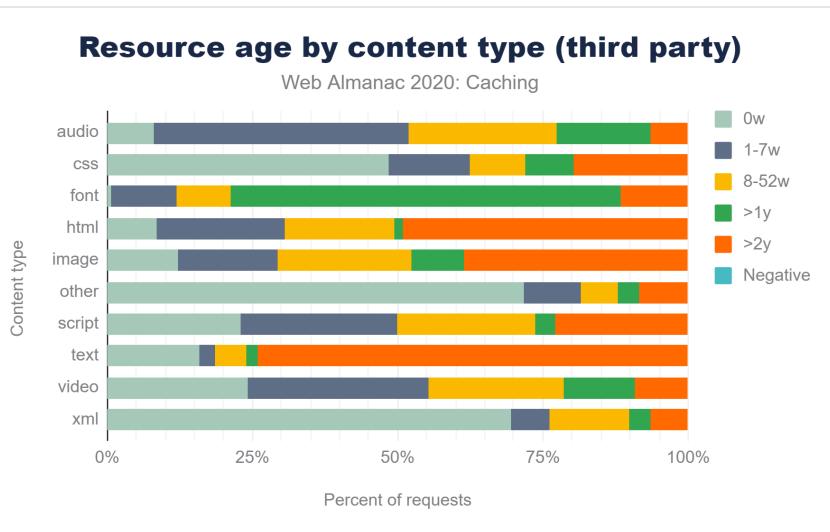


Figure 20.22. Resource age by Content Type (3rd Party).

By comparing a resource's cacheability to its age, we can determine if the TTL is appropriate or too low.

For example, the resource served below on 18 Oct 2020 was last modified on 30 Aug 2020, which means that it was well over a month old at the time of delivery - this indicates that it is an object which does not change frequently. However, the `Cache-Control` header says that the browser can cache it for only 86,400 seconds (one day). This is a case where a longer TTL might be appropriate, to avoid the browser needing to re-request it (even conditionally) - especially if the website is one that a user might visit multiple times over the course of several days.

```

> HTTP/1.1 200
> Date: Sun, 18 Oct 2020 19:36:57 GMT
> Content-Type: text/html; charset=utf-8
> Content-Length: 3052
> Vary: Accept-Encoding
> Last-Modified: Sun, 30 Aug 2020 16:00:30 GMT
> Cache-Control: public, max-age=86400

```

Overall, 60.7% of resources served on the web have a cache TTL that could be considered too short compared to its content age. Furthermore, the median delta between the TTL and age is 25 days - again, an indication of significant under-caching.

When we break this out by first party vs third party in the following table, we can see that more than two-thirds (61.6%) of first-party resources can benefit from a longer TTL. This clearly highlights a need to spend extra attention focusing on what is cacheable, and then ensuring that caching is configured correctly.

<i>Client</i>	<i>1st party</i>	<i>3rd party</i>	<i>Overall</i>
desktop	61.6%	59.3%	60.7%
mobile	61.8%	57.9%	60.2%

Figure 20.23. Percent of requests with short TTLs.

## Identifying caching opportunities

Google's Lighthouse tool enables users to run a series of audits against web pages, and the cache policy audit evaluates whether a site can benefit from additional caching. It does this by comparing the content age (via the `Last-Modified` header) to the cache TTL and estimating the probability that the resource would be served from cache. Depending on the score, you may see a caching recommendation in the results, with a list of specific resources that could be cached.

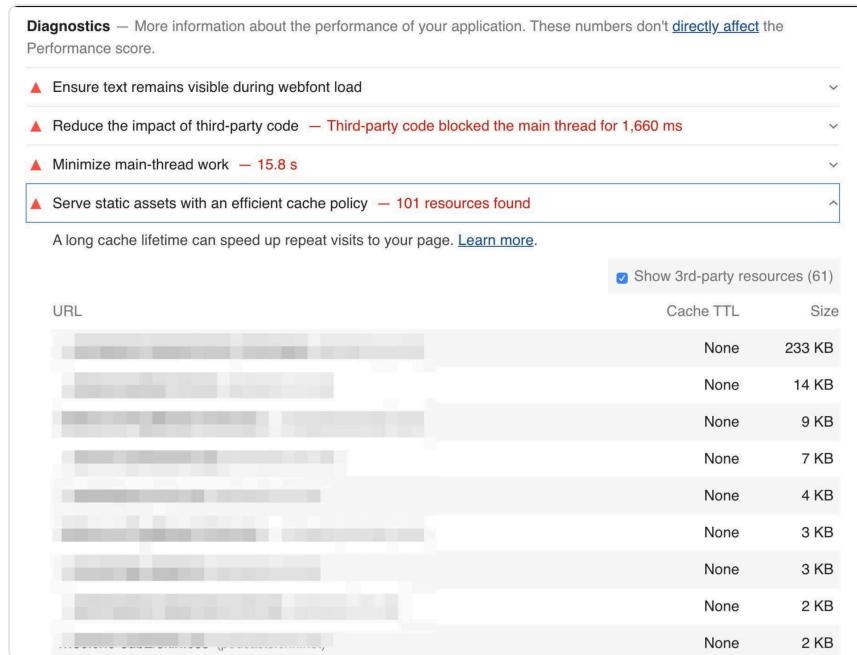


Figure 20.24. Lighthouse report highlighting potential cache policy improvements.

Lighthouse computes a score for each audit, ranging from 0% to 100%, and those scores are then factored into the overall scores. The caching score is based on potential byte savings. When we examine the Lighthouse results, we can get a perspective of how many sites are doing well with their cache policies.

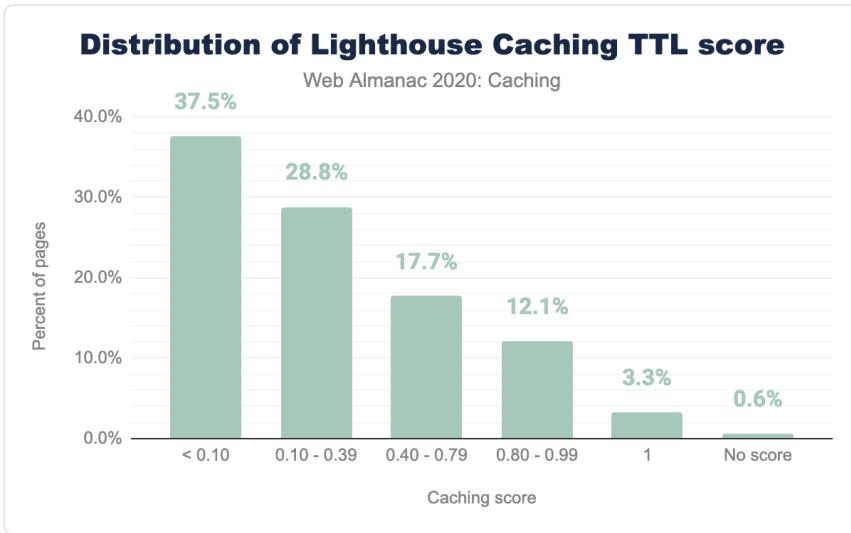


Figure 20.25. Distribution of Lighthouse audit scores for the `uses-long-cache-ttl` for mobile web pages.

Only 3.3% of sites scored a 100%, meaning that the vast majority of sites can benefit from some cache optimizations. Approximately two-thirds of sites score below 40%, with almost one-third of sites scoring less than 10%. Based on this, there is a significant amount of under-caching, resulting in excess requests and bytes being served across the network.

Lighthouse also indicates how many bytes could be saved on repeat views by enabling a longer cache policy. Of the sites that could benefit from additional caching, 78.6% of them can reduce their page weight by up to 2MB!

## Distribution of potential byte savings from caching

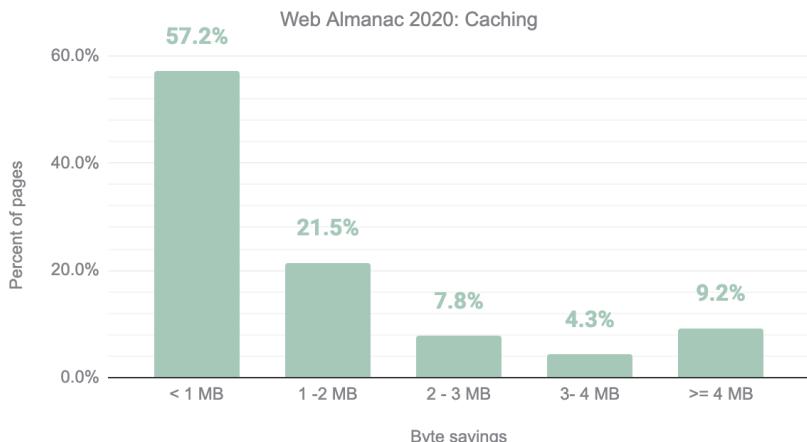


Figure 20.26. Distribution of potential byte savings from the Lighthouse caching audit.

## Conclusion

Caching is an incredibly powerful feature that allows browsers, proxies and other intermediaries (such as CDNs) to store web content and serve it to end users. The performance benefits of this are significant, since it reduces round trip times and minimizes costly network requests.

Caching is also a very complex topic, and one that is often left until late in the development cycle (due to requirements to see the very latest version of a site while it is still being designed), then being added in at the last minute. Additionally, caching rules are often defined once and then never changed, even as the underlying content on a site changes. Frequently a default value is chosen without careful consideration.

To correctly cache objects, there are numerous HTTP response headers that can convey freshness as well as validate cached entries, and `Cache-Control` directives provide a tremendous amount of flexibility and control.

Many object types and content that are typically considered to be uncacheable can actually be cached (remember: *cache as much as you can!*) and many objects are cached for too short a period of time, requiring repeated requests and revalidation (remember: cache for as long as you can!). However, website developers should be cautious about the additional opportunities for mistakes that come with over-caching content.

If the site is intended to be served through a CDN, additional opportunities for caching at the CDN to reduce server load and provide faster response to end-users should be considered, along with the related risks of accidentally caching private information.

However, 'powerful' and 'complex' do not imply 'difficult' - like most everything else, caching is controlled by rules which can be defined fairly easily to provide the best mix of cacheability and privacy. Regularly auditing your site to ensure that cacheable resources are cached appropriately is recommended, and tools like Lighthouse and REDbot do an excellent job of helping to simplify such an analysis.

## Authors

---



Rory Hewitt

roryhewitt

Enterprise Architect at Akamai<sup>41</sup>. Passionate about performance.



Raghu Ramakrishnan

raghuramakrishnan71

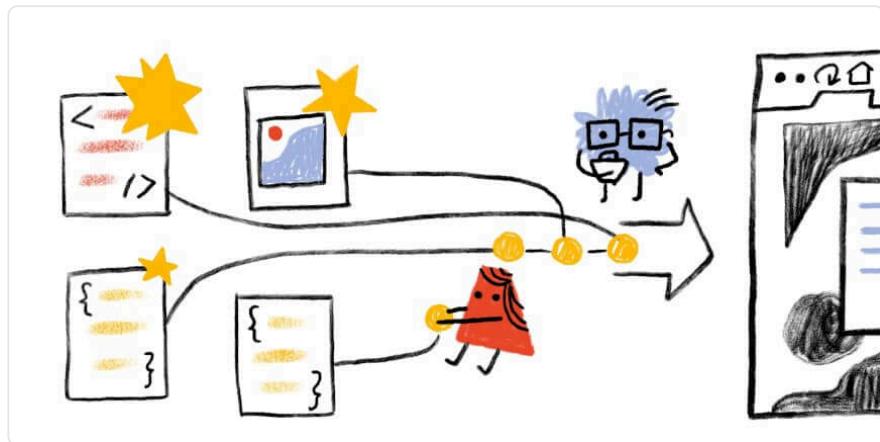
---

41. <https://www.akamai.com/>



# Part IV Chapter 21

# Resource Hints [UNEDITED]



Written by Leonardo Zizzamia

Reviewed by Jessica Nicolet, Patrick Meenan, Giovanni Punti, Minko Gechev, and notwillk

Analyzed by Katie Hempenius

## Introduction

Over the past decade resource hints have become essential primitives that allow developers to improve page performance and therefore the user experience.

Preloading resources and having browsers apply some intelligent prioritization is something that was actually started way back in 2009 by IE8 with something called the preloader. In addition to the HTML parser, IE8 had a lightweight look-ahead preloader that scanned for tags that could initiate network requests (`<script>`, `<link>`, and `<img>`).

Over the following years, browser vendors did more and more of the heavy lifting, each adding their own special sauce for how to prioritize resources. But it's important to understand that the browser alone has some limitations. As developers however, we can overcome these limits by making good use of resource hints and help decide how to prioritize resources, determining which should be fetched or preprocessed to further boost page performance.

In particular we can mention a few of the victories resource hints achieved/made in the last

year:

- CSS-Tricks web fonts showing up faster on a 3G first render
- wix.com using resource hints got 10% improvement for FCP
- Ironmongerydirect.co.uk by using preconnect improved product image loading by 400ms at the median and greater than 1s at 95th percentile
- Facebook.com used preload for faster navigation

Let's take a look at most predominant resource hints supported by most browsers today: `dns-prefetch`, `preconnect`, `preload`, `prefetch` and native lazy loading.

When used with each individual hint we advise to always measure the impact before and after in the field, by using libraries like WebVitals, Perfume.js or any other utility that supports the Web Vitals metrics.

## **dns-prefetch**

`dns-prefetch` helps resolve the IP address for a given domain ahead of time. As the oldest resource hint available, it uses minimal CPU and network resources compared to `preconnect`, and helps the browser to avoid experiencing the "worst-case" delay for DNS resolution, which can be over 1 second.

```
<link rel="dns-prefetch" href="https://www.googletagmanager.com/">
```

Be mindful when using `dns-prefetch` as even if they are lightweight to do it's easy to exhaust browser limits for the number of concurrent in-flight DNS requests allowed (Chrome still has a limit of 6).

## **preconnect**

`preconnect` helps resolve the IP address and open a TCP/TLS connection for a given domain ahead of time. Similar to `dns-prefetch` it is used for any cross-origin domain and helps the browser to warm up any resources used during the initial page load.

```
<link rel="preconnect" href="https://www.googletagmanager.com/">
```

Be mindful when you use `preconnect`:

- Only warm up the most frequent and significant resources.

- Avoid warm up origins used too late in the initial load.
- Use it for no more than three origins because it can have CPU and battery cost.

Lastly, `preconnect` is not available for Internet Explorer or Firefox, and using `dns-prefetch` as a fallback is highly advised.

## **preload**

The `preload` hint initiates an early request. This is useful for loading important resources that would otherwise be discovered late by the parser.

```
<link rel="preload" href="style.css" as="style">
<link rel="preload" href="main.js" as="script">
```

Be mindful of what you are going to `preload`, because it can delay the download of other resources, so use it only for what is most critical to help you improve the Largest Contentful Paint (LCP). Also, when used on Chrome, it tends to over-prioritize `preload` resources and potentially dispatches preloads before other critical resources.

Lastly, if used in a HTTP response header, some CDN's will also automatically turn a `preload` into a HTTP/2 push which can over-push cached resources.

## **prefetch**

The `prefetch` hint allows us to initiate low-priority requests we expect to be used on the next navigation. The Hint will download the resources and drop it into the HTTP cache for later usage. Important to notice, `prefetch` will not execute or otherwise process the resource, and to execute it the page will still need to call the resource by the `<script>` tag.

```
<link rel="prefetch" as="script" href="next-page.bundle.js">
```

There are a variety of ways to implement the resources predictions logic, could be based on signals like user mouse movement, common user flows/journeys or even based on a combination of both on top of Machine Learning.

Be mindful, depending on the quality of HTTP/2 prioritization of the CDN used, `prefetch` prioritization could either improve performance or make it slower, by over prioritizing `prefetch` requests and taking away important bandwidth for the initial load. Make sure to double check the CDN you are using and adapt to take into consideration some of the best

practices shared in Andy Davies's article.

## Native Lazy loading

The native lazy loading hint is a native browser API for deferring the load of offscreen images and iframes. By using it, assets that are not needed during the initial page load will not initiate a network request, this will reduce data consumption and improve page performance.

```

```

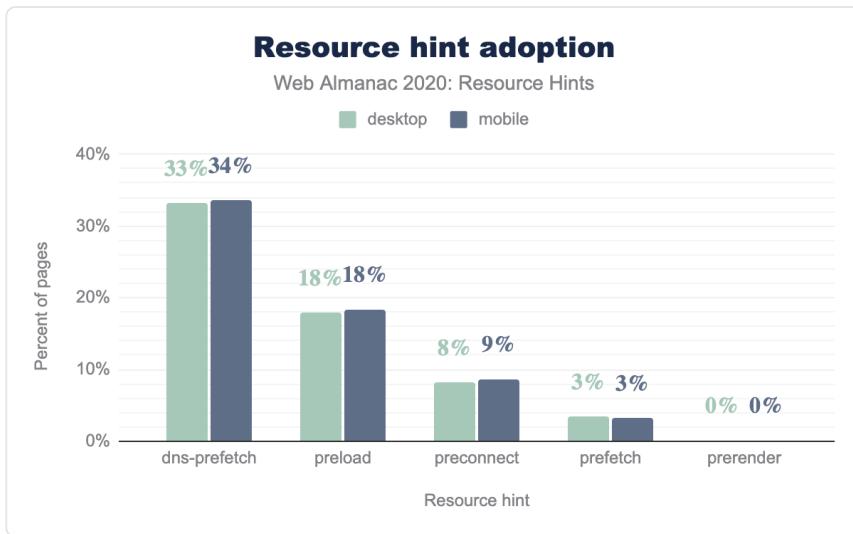
Be mindful Chromium's implementation of lazy-loading thresholds logic historically has been quite conservative, keeping the offscreen limit to 3000px. During the last year the limit has been actively tested and improved on to better align developer expectations, and ultimately moving the thresholds to 1250px. Also, there is no standard across the browsers and no ability for web developers to override the defaults thresholds provided by the browsers, yet.

## Resource Hints

Based on the HTTP Archive, let's jump into analyzing the 2020 trends, and compare the data with the previous 2019 dataset.

### Hints adoption

More and more web pages are using the main resource hints, and in 2020 we are seeing the adoption remains consistent between desktop & mobile.



*Figure 21.1. Adoption of resource hints*

The relative popularity of `dns-prefetch` with 33% adoption compared with other resource hints is unsurprising as it first appeared in 2009, and has the widest support out of all major resource hints.

Compared to 2019 the `dns-prefetch` had a 4% increase in Desktop adoption. We saw a similar increase for `preconnect` as well. One key reason this was the largest growth between all hints, is the clear and useful advice the Lighthouse audit is giving on this matter](<https://web.dev/uses-rel-preconnect/>). Starting from this year's report we also introduce how the latest dataset performs against Lighthouse recommendations.

`preload` usage has had a slower growth with only a 2% increase from 2019. This could be in part because it requires a bit more attention. While you only need the domain to use `dns-prefetch` and `preconnect`, you must specify the resource to use `preload`. While `dns-prefetch` and `preconnect` are reasonably low risk—though still can be abused—`preload` has a much greater potential to actually damage performance if used incorrectly.

`prefetch` is used by 3% of sites on Desktop, making it the least widely used resource hint. This low usage may be explained by the fact that `prefetch` is useful for improving subsequent—rather than current—page loads. Thus, it will be overlooked if a site is only focused on improving its landing page, or the performance of the first page viewed. In the coming years with a more clear definition on what to measure for improving subsequent page experience, it could help teams prioritize `prefetch` adoption with clear performance quality goals to reach.

## Hints per page

Across the board developers are learning how to better use resource hints, and compared to 2019 we've seen an improved use of `preload`, `prefetch` and `preconnect`. For expensive operations like `preload` and `preconnect` the median usage on desktop decreased from 2 to 1. We have seen the opposite for loading future resources with a lower priority with `prefetch`, with an increase from 1 to 2 in median per page.

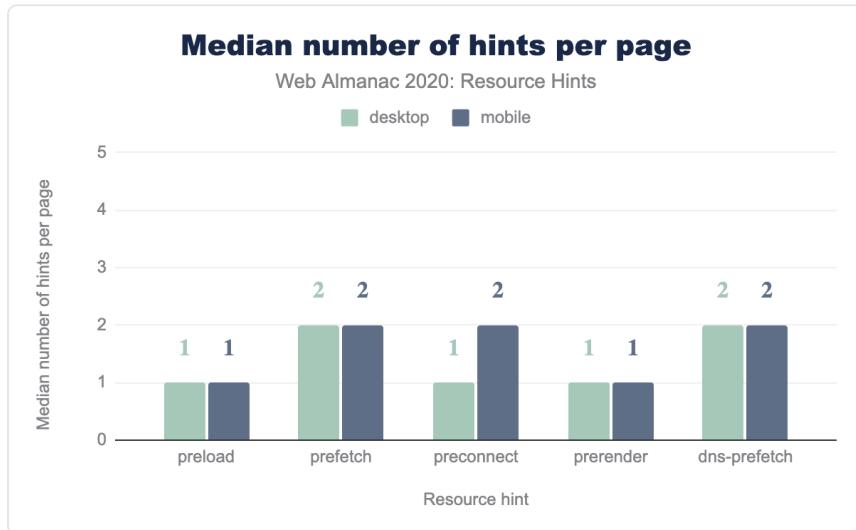


Figure 21.2. Median number of hints per page.

Resource hints are most effective when they're used selectively ("when everything is important, nothing is"). Having a more clear definition of what resources help improve critical rendering, versus future navigation optimizations, can move the focus away from using `preconnect` and more towards `prefetch` by shifting some of the resource prioritization and freeing up bandwidth for what most helps the user at first.

However, this hasn't stopped some misuse of the `preload` hint, since in one instance we discovered a page dynamically adding the hint and causing an infinite loop that created over 20k new preloads.

A large, bold, blue number "20,931" is displayed in a sans-serif font.

Figure 21.3. The most preload hints per page 🤯

As we create more and more automation with resource hints, be cautious when dynamically injecting preload hints - or any elements for that matter!

### The `as` attribute

With `preload` and `prefetch`, it's crucial to use the `as` attribute to help the browser prioritize the resource more accurately. Doing so allows for proper storage in the cache for future requests, applying the correct Content Security Policy (CSP), and setting the correct `Accept` request headers.

With `preload` many different content-types can be preloaded and the full list follows the recommendations made in the Fetch spec. The most popular is the `script` type with 64% usage.

A large, bold, blue percentage "64%" is displayed in a sans-serif font.

Figure 21.4. The percent of preload hints on mobile using the `scripts` type.

This is likely related to a large group of sites built as Single Page Apps that need the main bundle as soon as possible to start downloading the rest of their JS dependencies. Subsequent usage comes from font at 8%, style at 5%, image at 1%, and fetch at 1%.

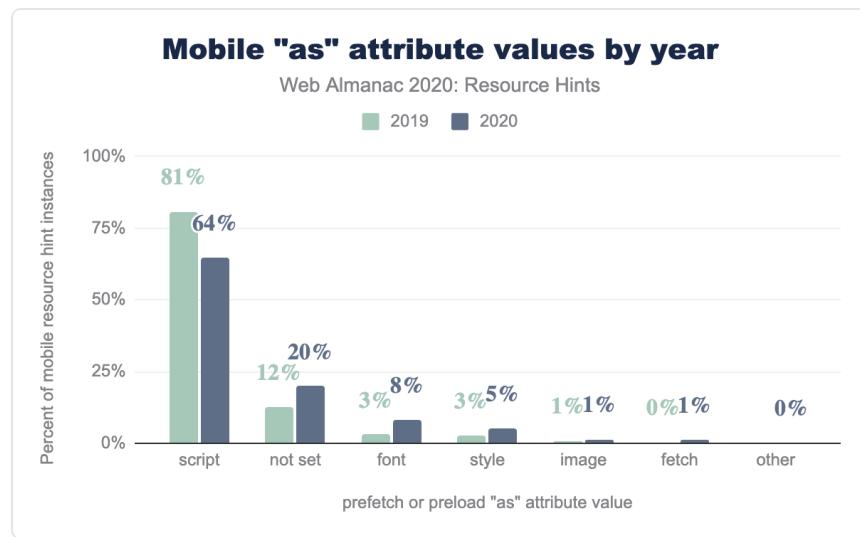


Figure 21.5. Mobile "as" attribute values by year.

Compared to the trend in 2019, we've seen rapid growth in font and style usage with the `as` attribute. This is likely related to both developers increasing the priority of critical CSS and also combining `preload` `fonts` with `display:optional` to improve Cumulative Layout Shift (CLS).

Be mindful that omitting the `as` attribute, or having an invalid value will make it harder for the browser to determine the correct priority and in some cases, such as scripts, can even cause the resource to be fetched twice.

## The `crossorigin` attribute

With `preload` and `preconnect` resources that have CORS enabled, such as fonts, it's important to include the `crossorigin` attribute, in order for the resource to be properly used. If the `crossorigin` attribute is absent, the request will follow the single-origin policy thereby making the use of `preload` useless.

The latest trend indicates that by using `preload` with the `crossorigin` attribute we have 34% anonymous (or equivalent) cases, and 0.43% use-credentials cases. This has evolved in conjunction with the increase in font-preloading mentioned earlier.

```
<link rel="preload" href="ComicSans.woff2" as="font" type="font/woff2" crossorigin>
```

Be mindful that fonts preloaded without the `crossorigin` attribute will be fetched twice!

## The `media` attribute

When it's time to choose a resource for use with different screen sizes, reach for the `media` attribute with `preload` to optimize your media queries.

```
<link rel="preload" href="desktop.css" as="style" media="only screen and (max-width: 600px)">
<link rel="preload" href="mobile.css" as="style" media="all and (max-width: 600px)">
```

Seeing over 2,100 different combinations of media queries in the 2020 dataset encourages us to consider how wide the variance is between concept and implementation of responsive design from site to site. The ever popular `767px/768px` breakpoints (as popularised by Bootstrap amongst others) can be seen in the data.

## Best practices

Using resource hints can be confusing at times, so let's go over some quick best practices to follow based on Lighthouse's automated audit.

To safely implement `dns-prefetch` and `preconnect` make sure to have them in separate `link` tags.

```
<link rel="preconnect" href="http://example.com">
<link rel="dns-prefetch" href="http://example.com">
```

Implementing a `dns-prefetch` fallback in the same `link` tag causes a bug in Safari that cancels the `preconnect` request.

1.93%

*Figure 21.6. Resource hints on desktop that use either `preconnect` or `dns-prefetch` use both in the same hint.*

Close to 2% of pages (~40k) reported the issue of both `preconnect` & `dns-prefetch` in a

single resource.



*Figure 21.7. Pages that pass the preconnect Lighthouse audit*

We saw only 19.67% of pages passing Lighthouse's "Preconnect to required origins" audit, creating a large opportunity for thousands of websites to start using `preconnect` or `dns-prefetch` to establish early connections to important third-party origins.



*Figure 21.8. Pages that pass the preload Lighthouse audit*

Running Lighthouse's "Preload key requests" audit resulted in 84.6% of pages passing the test, which is an astonishing result. If you are looking to use `preload` for the first time, remember, fonts and critical scripts are a good place to start.

## Native Lazy Loading

Now let's celebrate the first year of the Native Lazy Loading API, which at the time of publishing already has over 72% browser support. This new API can be used to defer the load of below-the-fold iframes and images on the page until the user scrolls near them. This can reduce data usage, memory usage, and helps speed up above-the-fold content. Opting-in to lazy load is as simple as adding `loading=lazy` on `<iframe>` or `<img>` elements.



*Figure 21.9. Percentage of pages using native lazy loading set to "lazy"*

Adoption is still in its early days, especially with the official thresholds earlier this year being too conservative, and only recently aligning with developer expectations. With almost 72% of

browsers supporting native image/source lazy loading, this is another area of opportunity especially for pages looking to improve data usage and performance on low-end devices.

A large, bold, blue percentage value '68.65%' is displayed prominently, likely representing the percentage of pages passing the offscreen images Lighthouse audit.

Figure 21.10. Percentage of pages passing the offscreen images Lighthouse audit

Running Lighthouse's "Defer offscreen images" audit resulted in 68.65% of pages passing the test. For those pages there is an opportunity to lazy-load images after all critical resources have finished loading.

Be mindful to run the audit on both desktop and mobile as images may move off screen when the viewport changes.

## Predictive prefetching

Combining `prefetch` with Machine Learning can help with performance improvement of the subsequent page(s). One solution is Guess.js which made the initial breakthrough in predictive-prefetching, with over a dozen websites already using it in production.

Predictive prefetching is a technique that uses methods from data analytics and machine learning to provide a data-driven approach. Guess.js is a library that has predictive prefetching support for popular frameworks (Angular, Nuxt.js, Gatsby, and Next.js) and you can take advantage of it today. It ranks the possible navigations from a page and prefetches only the JavaScript that is likely to be needed next.

Depending on the training set, the prefetching of Guess.js comes with over 90% accuracy.

Overall, predictive prefetching is still uncharted territory but combined with prefetching on mouse over and even a Service Worker strategy, it has great potential to provide instant experiences for all users of the website, while saving their data.

## HTTP/2 Push

HTTP/2 has a feature called server push that can potentially improve page performance when your product experiences long RTTs or server processing. In brief, rather than waiting for the client to send a request, the server preemptively pushes a resource that it predicts the client will request soon afterwards.



Figure 21.11. Percentage of HTTP/2 Push pages using preload/nopush

HTTP/2 Push is often initiated through the `preload` link header. In the 2020 dataset we have seen 1% of mobile pages using HTTP/2 Push, and of those 75% of preload header links use the `nopush` option in the page request. This means that even though a website is using the `preload` resource hint, the majority prefer to use just this and disable HTTP/2 pushing of that resource.

It's important to mention that HTTP/2 Push can also damage performance if not used correctly which probably explains why it is often disabled.

One solution to this, is to use the PRPL Pattern which stands for **P**ush (or `preload`) the critical resources, **R**ender the initial route as soon as possible, **P**re-cache remaining assets, and **L**azy-load other routes and non-critical assets. This is possible only if your website is a Progressive Web App and uses a Service Worker to improve the caching strategy. By doing this all subsequent requests never even go out to the network, and so there's no need to push all the time and we still get the best of both worlds.

## Service Workers

For both `preload` and `prefetch` we've had an increase in adoption when the page is controlled by a Service Worker. This is because of the potential to both improve the resource prioritization by preloading when the Service Worker is not active yet and intelligently prefetching future resources while letting the Service Worker cache them before they're needed by the user.

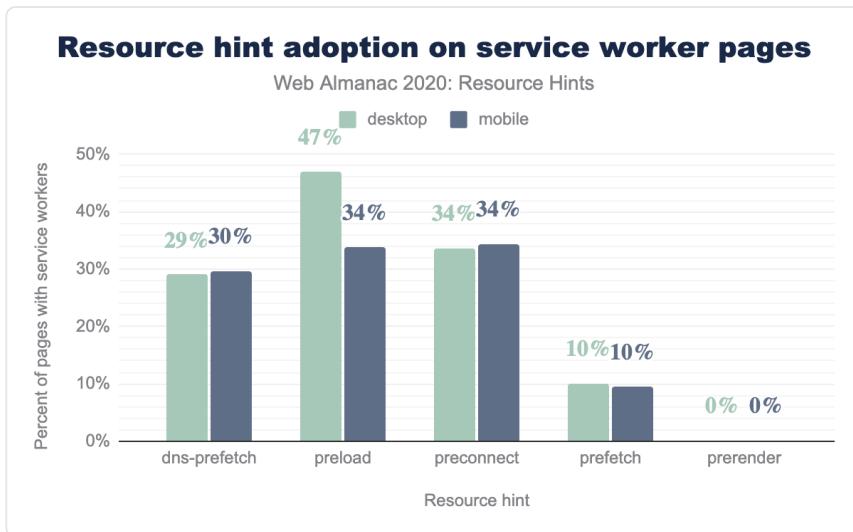


Figure 21.12. Resource hint adoption on service worker pages.

For `preload` on desktop we have an outstanding 47% rate of adoption and `prefetch` a 10% rate of adoption. In both cases the data is much higher compared to average adoption without a Service Worker.

As mentioned earlier, the PRPL Pattern will play a significant role in the coming years in how we combine resource hints with the Service Worker caching strategy.

## Future

Let's dive into a couple of experimental hints. Very close to release we have Priority Hints, at the moment actively experimented with the web community. Lastly we have the 103 Early Hints in HTTP/2, which is still in early inception and there are a few players like [Chrome and Fastly collaborating for upcoming test trials]](<https://www.fastly.com/blog/beyond-server-push-experimenting-with-the-103-early-hints-status-code>).

## Priority Hints

Priority hints are an API for expressing the fetch priority of a resource: high, low, or auto. They can be used to help de-prioritize images (e.g. inside a Carousel), re-prioritize scripts and even help de-prioritize fetches.

This new hint can be used either as an HTML tag or by changing the priority of fetch requests

via the `importance` option, which takes the same values as the HTML attribute.

With `preload` and `prefetch`, the priority is set by the browser depending on the type of resource. By using Priority Hints we can force the browser to change the default option.



*Figure 21.13. The rate of priority hint adoption on mobile*

So far only 0.77% websites adopted this new hint as Chrome is still actively experimenting, and at the time of this article's release the feature is on-hold.



*Figure 21.14. Priority hints on mobile are on script elements*

The largest use is with script elements, which is unsurprising as the number of JS primary and third-party files continues to grow.



*Figure 21.15. Priority hints on mobile have "low" importance*

There are over 79% of resources with "high" priority, but something we should pay even more attention to is the 16% of resources with "low" priority. Priority Hints have a clear advantage as a defense mechanism rather than an offense, by helping the browser decide what to de-prioritize and giving back significant CPU and Bandwidth to complete critical requests first.

## 103 Early Hints in HTTP/2

Previously we mentioned that HTTP/2 Push could actually cause regression in cases where assets being pushed were already in the browser cache. The 103 Early Hints proposal aims to provide similar benefits promised by HTTP/2 push. With an architecture that is potentially 10x

simpler, it addresses the long RTT's or server processing without suffering from the known worst-case issue of unnecessary round trips with server push.

As of right now you can follow the conversation on Chromium with issues 671310, 1093693 and 1096414.

## Conclusion

During the past year resource hints increased in adoption, and they have become essential APIs for developers to have more granular control over many aspects of resource prioritizations and ultimately, user experience. But let's not forget that these are hints, not instructions and unfortunately the Browser and the network will always have the final say.

Sure, you can slap them on a bunch of elements, and the browser may do what you're asking it to. Or it may ignore some hints and decide the default priority is the best choice for the given situation. In any case, make sure to have a playbook for how to best use these hints:

- Identify key pages for the user experience.
- Analyze the most important resources to optimize.
- Adopt the PRPL Pattern when possible.
- Measure the performance experience before and after each implementation.

As a final note, let's remember that the web is for everyone. We must continue to protect it and stay focused on building experiences that are easy and frictionless.

We are thrilled to see that year after year we get incrementally closer to offering all the APIs required to simplify building a great web experience for everyone, and we can't wait to see what comes next.

## Author



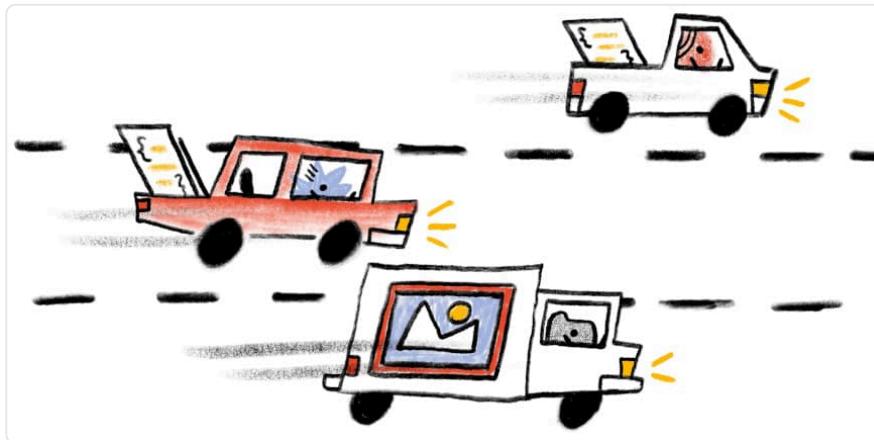
Leonardo Zizzamia

@Zizzamia   Zizzamia   <https://twitter.com/zizzamia>



# Part IV Chapter 22

# HTTP/2



*Written by Andrew Galloni, Robin Marx, and Mike Bishop*

*Reviewed by Lucas Pardue, Barry Pollard, and Sawood Alam*

*Analyzed by Greg Wolf*

## Introduction

HTTP is an application layer protocol designed to transfer information between networked devices and runs on top of other layers of the network protocol stack. After HTTP/1 was released, it took over 20 years until the first major update, HTTP/2, was made a standard in 2015.

It didn't stop there: over the last four years, HTTP/3 and QUIC (a new latency-reducing, reliable, and secure transport protocol) have been under standards development in the IETF QUIC working group. There are actually two protocols that share the same name: "Google QUIC" ("gQUIC" for short), the original protocol that was designed and used by Google, and the newer IETF standardized version (IETF QUIC/QUIC). IETF QUIC was based on gQUIC, but has grown to be quite different in design and implementation. On October 21, 2020, draft 32 of IETF QUIC reached a significant milestone when it moved to Last Call. This is the part of the standardization process when the working group believes they are almost finished and requests a final review from the wider IETF community.

This chapter reviews the current state of HTTP/2 and gQUIC deployment, to establish how well some of the newer features of the protocol, such as prioritization and server push, have been adopted. We then look at the motivations for HTTP/3, describe the major differences between the protocol versions and discuss the potential challenges in upgrading to a UDP-based transport protocol with QUIC.

## HTTP/1.0 to HTTP/2

As the HTTP protocol has evolved, the semantics of HTTP have stayed the same, with no changes to the HTTP methods (such as GET or POST), status codes (200, or the dreaded 404), URLs, or header fields. Where the HTTP protocol has changed, the differences have been the wire-encoding and the use of features of the underlying transport.

HTTP/1.0, published in 1996, defined the text-based application protocol, allowing clients and servers to exchange messages in order to request resources. A new TCP connection was required for each request/response, which introduced overhead. TCP connections use a congestion control algorithm to maximise how much data can be in-flight. This process takes time for each new connection. This "slow-start" means that not all the available bandwidth is used immediately.

In 1997, HTTP/1.1 was introduced to allow TCP connection reuse by adding "keep-alives", aimed at reducing the total cost of connection start-ups. Over time, increasing website performance expectations led to the need for concurrency of requests. HTTP/1.1 could only request another resource after the previous response had completed. Therefore additional TCP connections had to be established, reducing the impact of the keep-alive connections and further increasing overhead.

HTTP/2, published in 2015, is a binary-based protocol that introduced the concept of bidirectional streams between client and server. Using these streams, a browser can make optimal use of a single TCP connection to *multiplex* multiple HTTP requests/responses concurrently. HTTP/2 also introduced a prioritization scheme to steer this multiplexing; clients can signal a request priority that allows more important resources to be sent ahead of others.

## HTTP/2 Adoption

The data used in this chapter is sourced from the HTTP Archive and tests over seven million websites with a Chrome browser. As with other chapters, the analysis is split by mobile and desktop websites. When the results between desktop and mobile are similar, statistics are presented from the mobile dataset. You can find more details on the Methodology page. When reviewing this data, please bear in mind that each website will receive equal weight regardless of the number of requests. We suggest you think of this more as investigating the trends across a broad range of active websites.

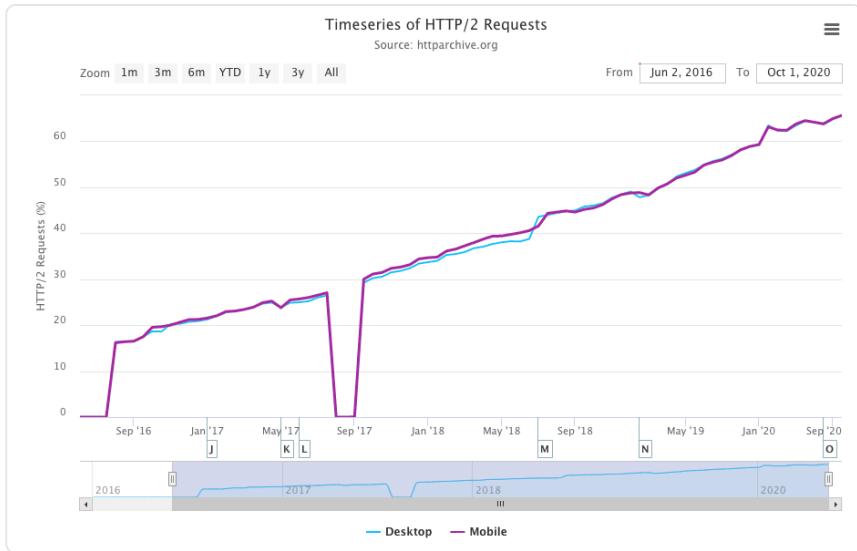


Figure 22.1. HTTP/2 usage by request. (Source: HTTP Archive)

Last year's analysis of HTTP Archive data showed that HTTP/2 was used for over 50% of requests and, as can be seen, linear growth has continued in 2020; now in excess of 60% of requests are served over HTTP/2.

# 64%

Figure 22.2. The percentage of requests that use HTTP/2.

When comparing Figure 22.3 with last year's results, there has been a **10% increase in HTTP/2 requests** and a corresponding 10% decrease in HTTP/1 requests. This is the first year that gQUIC can be seen in the dataset.

Protocol	Desktop	Mobile
HTTP/1.1	**34.47%	34.11%
HTTP/2	63.70%	63.80%
gQUIC	1.72%	1.71%

Figure 22.3. HTTP version usage by request.

*\*\* As with last year's crawl, around 4% of desktop requests did not report a protocol version. Analysis shows these to mostly be HTTP/1.1 and we worked to fix this gap in our statistics for future crawls and analysis. Although we base the data on the August 2020 crawl, we confirmed the fix in the October 2020 data set before publication which did indeed show these were HTTP/1.1 requests and so have added them to that statistic in above table.*

When reviewing the total number of website requests, there will be a bias towards common third-party domains. To get a better understanding of the HTTP/2 adoption by server install, we will look instead at the protocol used to serve the HTML from the home page of a site.

Last year around 37% of home pages were served over HTTP/2 and 63% over HTTP/1. This year, combining mobile and desktop, it is an equal 50% split with slightly more desktop sites being served over HTTP/2 for the first time, as shown in Figure 22.4.

Protocol	Desktop	Mobile
HTTP/1.0	0.06%	0.05%
HTTP/1.1	49.22%	50.05%
HTTP/2	49.97%	49.28%

Figure 22.4. HTTP version usage for home pages.

gQUIC is not seen in the home page data for two reasons. To measure a website over gQUIC, the HTTP Archive crawl would have to perform protocol negotiation via the alternative services header and then use this endpoint to load the site over gQUIC. This was not supported this year, but expect it to be available in next year's Web Almanac. Also, gQUIC is predominantly used for third-party Google tools rather than serving home pages.

The drive to increase security and privacy on the web has seen requests over TLS increase by over 150% in the last 4 years. Today, over 86% of all requests on mobile and desktop are encrypted. Looking only at home pages, the numbers are still an impressive 78.1% of desktop and 74.7% of mobile. This is important because HTTP/2 is only supported by browsers over TLS. The proportion served over HTTP/2, as shown in Figure 22.5, has also increased by 10 percentage points from last year, from 55% to 65%.

Protocol	Desktop	Mobile
HTTP/1.1	36.05%	34.04%
HTTP/2	63.95%	65.96%

Figure 22.5. HTTP version usage for HTTPS home pages.

With over 60% of websites being served over HTTP/2 or gQUIC, let's look a little deeper into the pattern of protocol distribution for all requests made across individual sites.

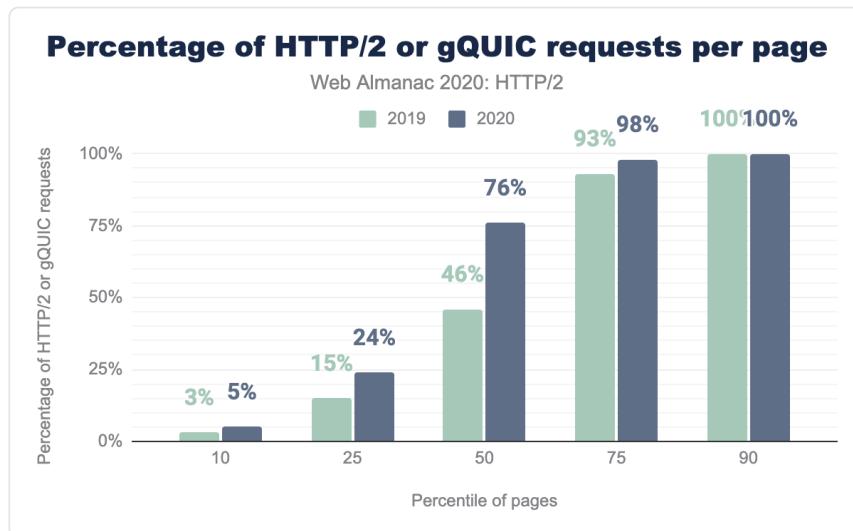
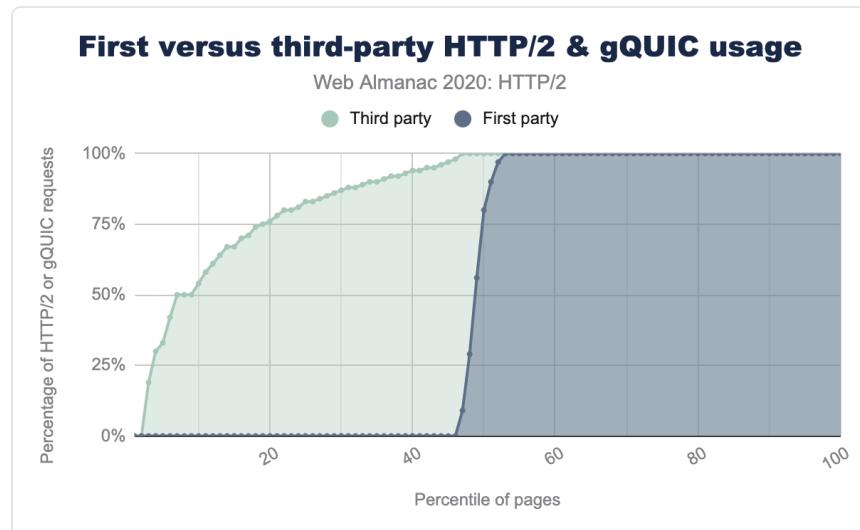


Figure 22.6. Compare the distribution of fraction of HTTP/2 requests per page in 2020 with 2019.

Figure 22.6 compares how much HTTP/2 or gQUIC is used on a website between this year and last year. The most noticeable change is that over half of sites now have 75% or more of their requests served over HTTP/2 or gQUIC compared to 46% last year. Less than 7% of sites make no HTTP/2 or gQUIC requests, while (only) 10% of sites are entirely HTTP/2 or gQUIC requests.

What about the breakdown of the page itself? We typically talk about the difference between first-party and third-party content. Third-party is defined as content not within the direct control of the site owner; providing functionality such as advertising, marketing or analytics. The definition of known third parties is taken from the third party web repository.

Figure 22.7 orders every website by the fraction of HTTP/2 requests for known third parties or first party requests compared to other requests. There is a noticeable difference as over 40% of all sites have no first-party HTTP/2 or gQUIC requests at all. By contrast, even the lowest 5% of pages have 30% of third-party content served over HTTP/2. This indicates that a large part of HTTP/2's broad adoption is driven by the third parties.



*Figure 22.7. The distribution of the fraction of third-party and first-party HTTP/2 requests per page.*

Is there any difference in which content-types are served over HTTP/2 or gQUIC? Figure 22.8 shows, for example, that 90% of websites serve 100% of third party fonts and audio over HTTP/2 or gQUIC, only 5% over HTTP/1 and 5% are a mix. The majority of third-party assets are either scripts or images, and are solely served over HTTP/2 or gQUIC on 60% and 70% of websites respectively.

## Third-party HTTP/2 or gQUIC usage by content-type

Web Almanac 2020: HTTP/2

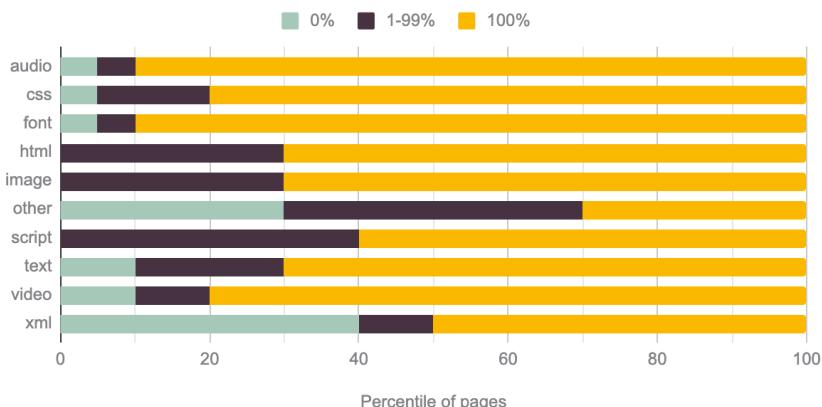


Figure 22.8. The fraction of known third-party HTTP/2 or gQUIC requests by content-type per website.

Ads, analytics, content delivery network (CDN) resources, and tag-managers are predominantly served over HTTP/2 or gQUIC as shown in Figure 22.9. Customer-success and marketing content is more likely to be served over HTTP/1.

## Third-party HTTP/2 or gQUIC usage by category

Web Almanac 2020: HTTP/2

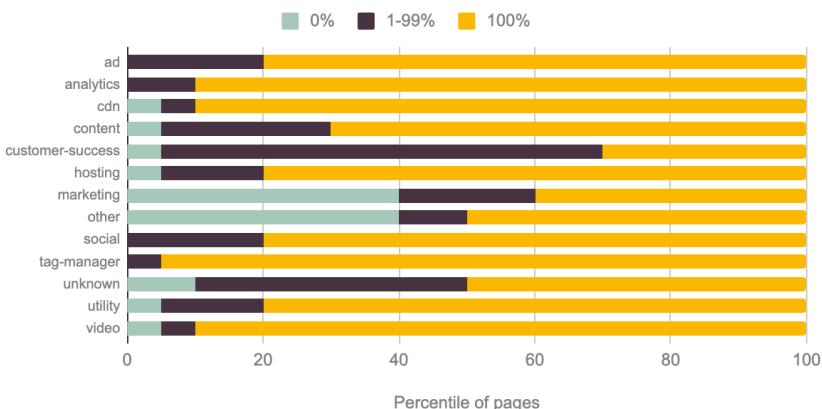


Figure 22.9. The fraction of known third-party HTTP/2 or gQUIC requests by category per website.

## Server support

Browser auto-update mechanisms are a driving factor for client-side adoption of new web standards. It's estimated that over 97% of global users support HTTP/2, up slightly from 95% measured last year.

Unfortunately, the upgrade path for servers is more difficult, especially with the requirement to support TLS. For mobile and desktop, we can see from Figure 22.10, that the majority of HTTP/2 sites are served by nginx, Cloudflare, and Apache. Almost half of the HTTP/1.1 sites are served by Apache.

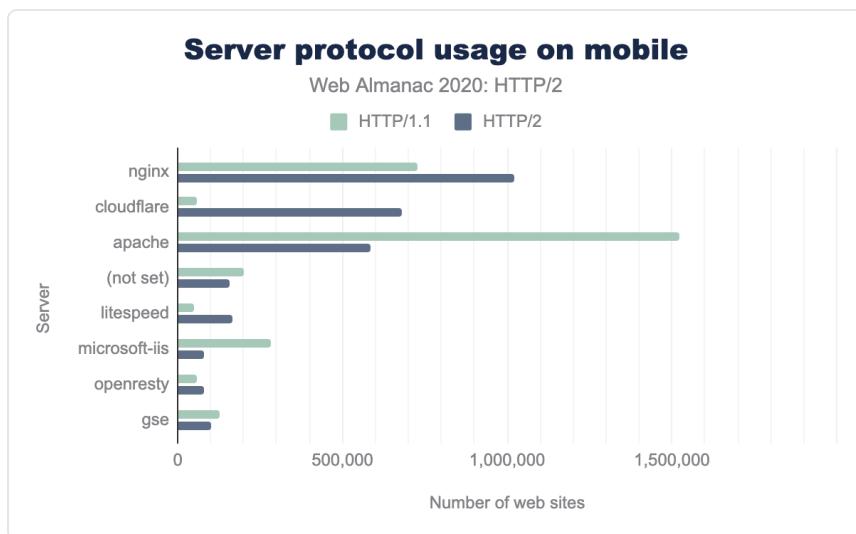


Figure 22.10. Server usage by HTTP protocol on mobile

How has HTTP/2 adoption changed in the last year for each server? Figure 22.11 shows a general HTTP/2 adoption increase of around 10% across all servers since last year. Apache and IIS are still under 25% HTTP/2. This suggests that either new servers tend to be nginx or it is seen as too difficult or not worthwhile to upgrade Apache or IIS to HTTP/2 and/or TLS.

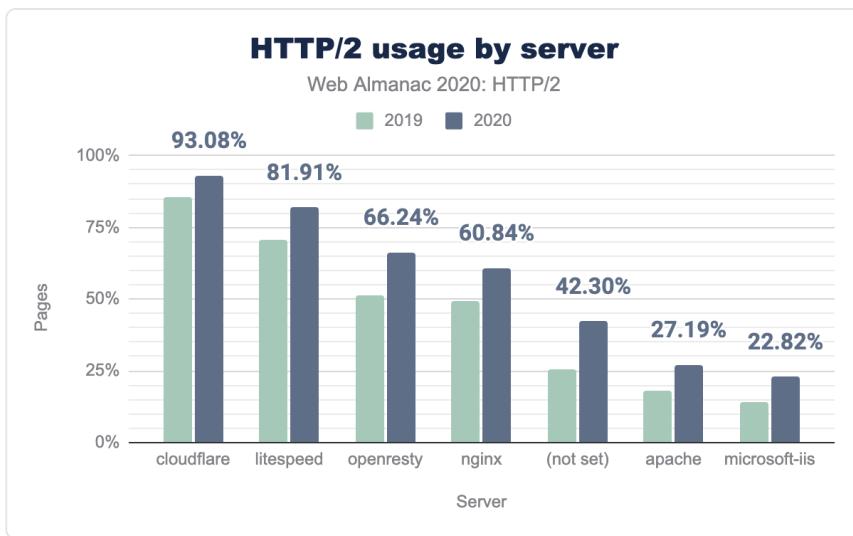


Figure 22.11. Percentage of pages served over HTTP/2 by sever

A long-term recommendation to improve website performance has been to use a CDN. The benefit is a reduction in latency by both serving content and terminating connections closer to the end user. This helps mitigate the rapid evolution in protocol deployment and the additional complexities in tuning servers and operating systems (see the Prioritization section for more details). To utilize the new protocols effectively, using a CDN can be seen as the recommended approach.

CDNs can be classed in two broad categories: those that serve the home page and/or asset subdomains, and those that are mainly used to serve third-party content. Examples of the first category are the larger generic CDNs (such as Cloudflare, Akamai, or Fastly) and the more specific (such as WordPress or Netlify). Looking at the difference in HTTP/2 adoption rates for home pages served with or without a CDN, we see:

- 80% of mobile home pages are served over HTTP/2 if a CDN is used
- 30% of mobile home pages are served over HTTP/2 if a CDN is not used

Figure 22.12 shows the more specific and the modern CDNs serve a higher proportion of traffic over HTTP/2.

<b>HTTP/2 (%)</b>	<b>CDN</b>
100%	Bison Grid, CDNsun, LeaseWeb CDN, NYI FTW, QUIC.cloud, Roast.io, Sirv CDN, Twitter, Zycada Networks
90 - 99%	Automattic, Azion, BitGravity, Facebook, KeyCDN, Microsoft Azure, NGENIX, Netlify, Yahoo, section.io, Airee, BunnyCDN, Cloudflare, GoCache, NetDNA, SFR, Sucuri Firewall
70 - 89%	Amazon CloudFront, BelugaCDN, CDN, CDN77, Erstream, Fastly, Highwinds, OVH CDN, Yottaa, Edgecast, Myra Security CDN, StackPath, XLabs Security
20 - 69%	Akamai, Aryaka, Google, Limelight, Rackspace, Incapsula, Level 3, Medianova, OnApp, Singular CDN, Vercel, CacheFly, Cedexis, Reflected Networks, Universal CDN, Yunjiasu, CDNetworks
< 20%	Rocket CDN, BO.LT, ChinaCache, KINX CDN, Zenedge, ChinaNetCenter

Figure 22.12. Percentage of HTTP/2 requests served by the first-party CDNs over mobile.

Types of content in the second category are typically shared resources (JavaScript or font CDNs), advertisements, or analytics. In all these cases, using a CDN will improve the performance and offload for the various SaaS solutions.

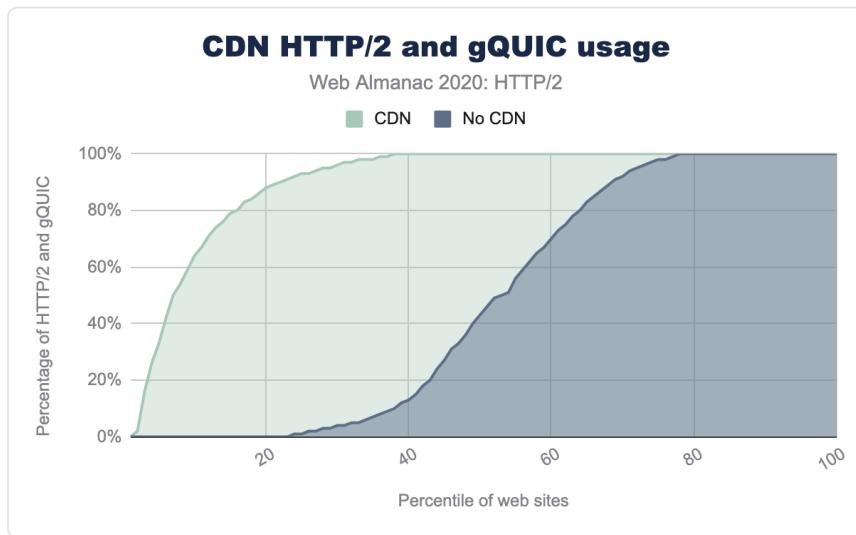


Figure 22.13. Comparison of HTTP/2 and gQUIC usage for websites using a CDN.

In Figure 22.13 we can see the stark difference in HTTP/2 and gQUIC adoption when a website is using a CDN. 70% of pages use HTTP/2 for all third-party requests when a CDN is used. Without a CDN, only 25% of pages use HTTP/2 for all third-party requests.

## HTTP/2 impact

Measuring the impact of how a protocol is performing is difficult with the current HTTP Archive approach. It would be really fascinating to be able to quantify the impact of concurrent connections, the effect of packet loss, and different congestion control mechanisms. To really compare performance, each website would have to be crawled over each protocol over different network conditions. What we can do instead is to look into the impact on the number of connections a website uses.

### Reducing connections

As discussed earlier, HTTP/1.1 only allows a single request at a time over a TCP connection. Most browsers get around this by allowing six parallel connections per host. The major improvement with HTTP/2 is that multiple requests can be multiplexed over a single TCP connection. This should reduce the total number of connections—and the associated time and resources—required to load a page.

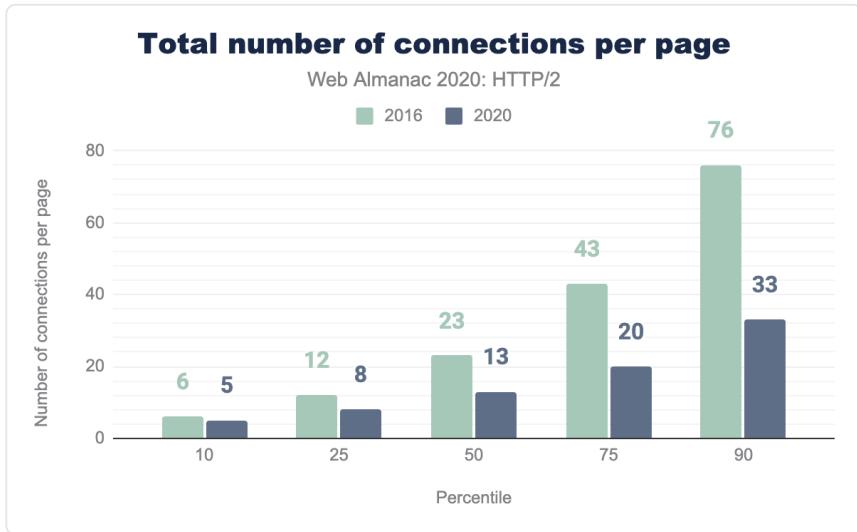


Figure 22.14. Distribution of total number of connections per page

Figure 22.15 shows how the number of TCP connections per page has reduced in 2020 compared with 2016. Half of all websites now use 13 or fewer TCP connections in 2020 compared with 23 connections in 2016; a 44% decrease. In the same time period the median number of requests has only dropped from 74 to 73. The median number of requests per TCP connection has increased from 3.2 to 5.6.

TCP was designed to maintain an average data flow that is both efficient and fair. Imagine a flow control process where each flow both exerts pressure on and is responsive to all other flows, to provide a fair share of the network. In a fair protocol, every TCP session does not crowd out any other session and over time will take  $1/N$  of the path capacity.

The majority of websites still open over 15 TCP connections. In HTTP/1.1, the six connections a browser could open to a domain can over time claim six times as much bandwidth as a single HTTP/2 connection. Over low capacity networks, this can slow down the delivery of content from the primary asset domains as the number of contending connections increases and takes bandwidth away from the important requests. This favors websites with a small number of third-party domains.

HTTP/2 does allow for connection reuse across different, but related domains. For a TLS resource, it requires a certificate that is valid for the host in the URI. This can be used to reduce the number of connections required for domains under the control of the site author.

## Prioritization

As HTTP/2 responses can be split into many individual frames, and as frames from multiple streams can be multiplexed, the order in which the frames are interleaved and delivered by the server becomes a critical performance consideration. A typical website consists of many different types of resources: the visible content (HTML, CSS, images), the application logic (JavaScript), ads, analytics for tracking site usage, and marketing tracking beacons. With knowledge of how a browser works, an optimal ordering of the resources can be defined that will result in the fastest user experience. The difference between optimal and non-optimal can be significant—as much as a 50% performance improvement or more!

HTTP/2 introduced the concept of prioritization to help the client communicate to the server how it thinks the multiplexing should be done. Every stream is assigned a weight (how much of the available bandwidth the stream should be allocated) and possibly a parent (another stream which should be delivered first). With the flexibility of HTTP/2's prioritization model, it is not altogether surprising that all of the current browser engines implemented different prioritization strategies, none of which are optimal.

There are also problems on the server side, leading to many servers either implementing prioritization poorly or not at all. In the case of HTTP/1.x, tuning the server-side send buffers to be as big as possible has no downside other than the increase in memory use (trading off memory for CPU) and is an effective way to increase the throughput of a web server. This is not true for HTTP/2, as data in the TCP send buffer cannot be re-prioritized if a request for a new, more important resource comes in. For an HTTP/2 server, the optimal send buffer size is thus the minimum amount of data required to fully utilize the available bandwidth. This allows the server to respond immediately if a higher-priority request is received.

This problem of large buffers messing with (re-)prioritization also exists in the network where it

goes by the name "bufferbloat". Network equipment would rather buffer packets than drop them when there's a short burst. However, if the server sends more data than the path to the client can consume, these buffers fill to capacity. These bytes already "stored" on the network limit the server's ability to send a higher-priority response earlier, just as a large send buffer does. To minimize the amount of data held in buffers, a recent congestion control algorithm such as BBR should be used.

This test suite maintained by Andy Davies measures and reports how various CDN and cloud hosting services perform. The bad news is that only 9 of the 36 services prioritize correctly. Figure 22.16 shows that for sites using a CDN, around 31.7% do not prioritize correctly. This is up from 26.82% last year, mainly due to the increase in Google CDN usage. Rather than relying on the browser-sent priorities, there are some servers that implement a server side prioritization scheme instead, improving upon the browser's hints with additional logic.

<b>CDN</b>	<b>Prioritize correctly</b>	<b>Desktop</b>	<b>Mobile</b>
Not using CDN	Unknown	59.47%	60.85%
Cloudflare	Pass	22.03%	21.32%
Google	Fail	8.26%	8.94%
Amazon CloudFront	Fail	2.64%	2.27%
Fastly	Pass	2.34%	1.78%
Akamai	Pass	1.31%	1.19%
Automattic	Pass	0.93%	1.05%
Sucuri Firewall	Fail	0.77%	0.63%
Incapsula	Fail	0.42%	0.34%
Netlify	Fail	0.27%	0.20%

Figure 22.15. HTTP/2 prioritization support in common CDNs.

For non-CDN usage, we expect the number of servers that correctly apply HTTP/2 prioritization to be considerably smaller. For example, NodeJS's HTTP/2 implementation does not support prioritization.

## Goodbye server push?

Server push was one of the additional features of HTTP/2 that caused some confusion and complexity to implement in practice. Push seeks to avoid waiting for a browser/client to

download a HTML page, parse that page, and only then discover that it requires additional resources (such as a stylesheet), which in turn have to be fetched and parsed to discover even more dependencies (such as fonts). All that work and round trips takes time. With server push, in theory, the server can just send multiple responses at once, avoiding the extra round trips.

Unfortunately, with TCP congestion control in play, the data transfer starts off so slowly that not all the assets can be pushed until multiple round trips have increased the transfer rate sufficiently. There are also implementation differences between browsers as the client processing model had not been fully agreed. For example, each browser has a different implementation of a *push cache*.

Another issue is that the server is not aware of resources the browser has already cached. When a server tries to push something that is unwanted, the client can send a `RST_STREAM` frame, but by the time this has happened, the server may well have already sent all the data. This wastes bandwidth and the server has lost the opportunity of immediately sending something that the browser actually did require. There were proposals to allow clients to inform the server of their cache status, but these suffered from privacy concerns.

As can be seen from the Figure 20.17 below, a very small percentage of sites use server push.

<b>Client</b>	<b>HTTP/2 pages</b>	<b>HTTP/2 (%)</b>	<b>gQUIC pages</b>	<b>gQUIC (%)</b>
Desktop	44,257	0.85%	204	0.04%
Mobile	62,849	1.06%	326	0.06%

Figure 22.16. Pages using HTTP/2 or gQUIC server push.

Looking further at the distributions for pushed assets in Figures 22.18 and 22.19, half of the sites push 4 or fewer resources with a total size of 140 KB on desktop and 3 or fewer resources with a size of 184 KB on mobile. For gQUIC, desktop is 7 or fewer and mobile 2. The worst offending page pushes 41 assets over gQUIC on desktop.

<b>Percentile</b>	<b>HTTP/2</b>	<b>Size (KB)</b>	<b>gQUIC</b>	<b>Size (KB)</b>
10	1	15.48	1	0.06
25	1	36.34	1	0.06
50	3	183.83	2	24.06
75	10	225.41	5	204.65
90	12	351.05	18	453.57

Figure 22.17. Distribution of pushed assets on desktop.

<b>Percentile</b>	<b>HTTP/2</b>	<b>Size (KB)</b>	<b>gQUIC</b>	<b>Size (KB)</b>
10	1	15.48	1	0.06
25	1	36.34	1	0.06
50	3	183.83	2	24.06
75	10	225.41	5	204.65
90	12	351.05	18	453.57

Figure 22.18. Distribution of pushed assets on mobile.

Looking at the frequency of push by content type in Figure 22.20, we see 90% of pages push scripts and 56% push CSS. This makes sense, as these can be small files typically on the critical path to render a page.

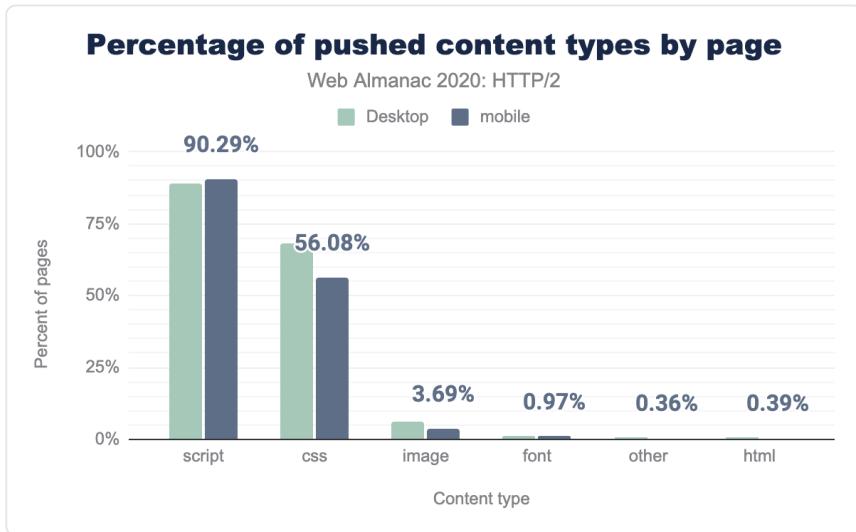


Figure 22.19. Percentage of pages pushing specific content types

Given the low adoption and after measuring how few of the pushed resources are actually useful (that is, they match a request that is not already cached), Google has announced the intent to remove push support from Chrome for both HTTP/2 and gQUIC. Chrome have also not implemented push for HTTP/3.

Despite all these problems, there are circumstances where server push can provide an improvement. The ideal use case is to be able to send a push promise much earlier than the HTML response itself. A scenario where this can benefit is when a CDN is in use. The "dead

time" between the CDN receiving the request and receiving a response from the origin can be used intelligently to warm up the TCP connection and push assets already cached at the CDN.

There was however no standardized method for how to signal to a CDN edge server that an asset should be pushed. Implementations instead reused the preload HTTP link header to indicate this. This simple approach appears elegant, but it does not utilize the dead time before the HTML is generated unless the headers are sent before the actual content is ready. It triggers the edge to push resources as the HTML is received at the edge, which will contend with the delivery of the HTML.

An alternative proposal being tested is RFC 8297, which defines an informative `HTTP/1.0 103 Early Hints` response. This permits headers to be sent immediately, without having to wait for the server to generate the full response headers. This can be used by an origin to suggest pushed resources to a CDN, or by a CDN to alert the client to resources that need to be fetched. However, at present, support for this from both a client and server perspective is very low, but growing.

## Getting to a better protocol

Let's say a client and server support both HTTP/1.1 and HTTP/2. How do they choose which one to use? The most common method is TLS Application Layer Protocol Negotiation (ALPN), in which clients send a list of protocols they support to the server, which picks the one it prefers to use for that connection. Because the browser needs to negotiate the TLS parameters as part of setting up an HTTPS connection, it can also negotiate the HTTP version at the same time. Since both HTTP/2 and HTTP/1.1 can be served from the same TCP port (443), browsers don't need to make this selection before opening a connection.

This doesn't work if the protocols aren't on the same port, use a different transport protocol (TCP versus QUIC), or if you're not using TLS. For those scenarios, you start with whatever is available on the first port you connect to, then discover other options. HTTP defines two mechanisms to change protocols for an origin after connecting: `Upgrade` and `Alt-Svc`.

### Upgrade

The `Upgrade` header has been part of HTTP for a long time. In HTTP/1.x, `Upgrade` allows a client to make a request using one protocol, but indicate its support for another protocol (like HTTP/2). If the server also supports the offered protocol, it responds with a status 101 (`Switching Protocols`) and proceeds to answer the request in the new protocol. If not, the server answers the request in HTTP/1.x. Servers can advertise their support of a different protocol using an `Upgrade` header on a response.

The most common application of `Upgrade` is WebSockets. HTTP/2 also defines an `Upgrade`

path, for use with its unencrypted cleartext mode. There is no support for this capability in web browsers however. Therefore, it's not surprising that less than 3% of HTTP/1.1 requests in our dataset received an `Upgrade` header in the response. A very small number of requests using TLS (0.0011% of HTTP/2, 0.064% of HTTP/1.1) also received `Upgrade` headers in response; these are likely cleartext HTTP/1.1 servers behind intermediaries which speak HTTP/2 and/or terminate TLS, but don't properly remove `Upgrade` headers.

## Alternative Services

Alternative Services (`Alt-Svc`) enables an HTTP origin to indicate other endpoints which serve the same content, possibly over different protocols. Although uncommon, HTTP/2 might be located at a different port or different host from a site's HTTP/1.1 service. More importantly, since HTTP/3 uses QUIC (hence UDP) where prior versions of HTTP use TCP, HTTP/3 will always be at a different endpoint from the HTTP/1.x and HTTP/2 service.

When using `Alt-Svc`, a client makes requests to the origin as normal. However, if the server includes a header or sends a frame containing a list of alternatives, the client can make a new connection to the other endpoint and use it for future requests to that origin.

Unsurprisingly, `Alt-Svc` usage is found almost entirely from services using advanced HTTP versions: 12.0% of HTTP/2 requests and 60.1% of gQUIC requests received an `Alt-Svc` header in response, as compared to 0.055% of HTTP/1.x requests. Note that our methodology here only captures `Alt-Svc` headers, not `ALTSVC` frames, so reality might be slightly understated.

While `Alt-Svc` can point to an entirely different host, support for this capability varies among browsers. Only 4.71% of `Alt-Svc` headers advertised an endpoint on a different hostname; these were almost universally (99.5%) advertising gQUIC and HTTP/3 support on Google Ads. Google Chrome ignores cross-host `Alt-Svc` advertisements for HTTP/2, so many of the other instances would have been ignored.

Given the rarity of support for cross-host HTTP/2, it's not surprising that there were virtually no (0.007%) advertisements for HTTP/2 endpoints using `Alt-Svc`. `Alt-Svc` was typically used to indicate support for HTTP/3 (74.6% of `Alt-Svc` headers) or gQUIC (38.7% of `Alt-Svc` headers).

## Looking toward the future: HTTP/3

HTTP/2 is a powerful protocol, which has found considerable adoption in just a few years. However, HTTP/3 over QUIC is already peeking around the corner! Over four years in the making, this next version of HTTP is almost standardized at the IETF (expected in the first half of 2021). At this time, there are already many QUIC and HTTP/3 implementations available,

both for servers and browsers. While these are relatively mature, they can still be said to be in an experimental state.

This is reflected by the usage numbers in the HTTP Archive data, where no HTTP/3 requests were identified at all. This might seem a bit strange, since Cloudflare has had experimental HTTP/3 support for some time, as have Google and Facebook. This is mainly because Chrome hadn't enabled the protocol by default when the data was collected.

However, even the numbers for the older gQUIC are relatively modest, accounting for less than 2% of requests overall. This is expected, since gQUIC was mostly deployed by Google and Akamai; other parties were waiting for IETF QUIC. As such, gQUIC is expected to be replaced entirely by HTTP/3 soon.

A large, bold, dark blue percentage sign reading 1.72%.

Figure 22.20. The percentage of requests that use gQUIC on desktop and mobile

It's important to note that this low adoption only reflects gQUIC and HTTP/3 usage for loading Web pages. For several years already, Facebook has had a much more extensive deployment of IETF QUIC and HTTP/3 for loading data inside of its native applications. QUIC and HTTP/3 already make up over 75% of their total internet traffic. It is clear that HTTP/3 is only just getting started!

Now you might wonder: if not everyone is already using HTTP/2, why would we need HTTP/3 so soon? What benefits can we expect in practice? Let's take a closer look at its internal mechanisms.

## QUIC and HTTP/3

Past attempts to deploy new transport protocols on the internet have proven difficult, for example Stream Control Transmission Protocol (SCTP). QUIC is a new transport protocol that runs on top of UDP. It provides similar features to TCP such as reliable in-order delivery and congestion control to prevent flooding the network.

As discussed in the HTTP/1.0 to HTTP/2 section, HTTP/2 *multiplexes* multiple different streams on top of one connection. TCP itself is woefully unaware of this fact, leading to inefficiencies or performance impact when packet loss or delays occur. More details on this problem, known as *head-of-line blocking* (HOL blocking), can be found [here](#).

QUIC solves the HOL blocking problem by bringing HTTP/2's streams down into the transport layer and performing per-stream loss detection and retransmission. So then we just put HTTP/2

over QUIC, right? Well, we've already mentioned some of the difficulties arising from having flow control in TCP and HTTP/2. Running two separate and competing streaming systems on top of each other would be an additional problem. Furthermore, because the QUIC streams are independent, it would mess with the strict ordering requirements HTTP/2 requires for several of its systems.

In the end, it was deemed easier to define a new HTTP version that uses QUIC directly and thus, HTTP/3 was born. Its high-level features are very similar to those we know from HTTP/2, but internal implementation mechanisms are quite different. HPACK header compression is replaced with QPACK, which allows manual tuning of the compression efficiency versus HOL blocking risk tradeoff, and the prioritization system is being replaced by a simpler one. The latter could also be back-ported to HTTP/2, possibly helping resolve the prioritization issues discussed earlier in this article. HTTP/3 continues to provide a server push mechanism, but Chrome, for example, does not implement it.

Another benefit of QUIC is that it is able to migrate connections and keep them alive even when the underlying network changes. A typical example is the so-called "parking lot problem". Imagine your smartphone is connected to the workplace Wi-Fi network and you've just started downloading a large file. As you leave Wi-Fi range, your phone automatically switches to the fancy new 5G cellular network. With plain old TCP, the connection would break and cause an interruption. But QUIC is smarter; it uses a *connection ID*, which is more robust to network changes, and provides an active *connection migration* feature for clients to switch without interruption.

Finally, TLS is already used to protect HTTP/1.1 and HTTP/2. QUIC, however, has a deep integration of TLS 1.3, protecting both HTTP/3 data and QUIC packet metadata, such as packet numbers. Using TLS in this way improves end-user privacy and security and makes continued protocol evolution easier. Combining the transport and cryptographic handshakes means that connection setup takes just a single RTT, compared to TCP's minimum of two and worst case of four. In some cases, QUIC can even go one step further and send HTTP data along with its very first message, which is called 0-RTT. These fast connection setup times are expected to really help HTTP/3 outperform HTTP/2.

### So, will HTTP/3 help?

On the surface, HTTP/3 is really not all that different from HTTP/2. It doesn't add any major features, but mainly changes how the existing ones work under the surface. The real improvements come from QUIC, which offers faster connection setups, increased robustness, and resilience to packet loss. As such, HTTP/3 is expected to do better than HTTP/2 on worse networks, while offering very similar performance on faster systems. However, that is if the web community can get HTTP/3 working, which can be challenging in practice.

## Deploying and discovering HTTP/3

Since QUIC and HTTP/3 run over UDP, things aren't as simple as with HTTP/1.1 or HTTP/2. Typically, an HTTP/3 client has to first discover that QUIC is available at the server. The recommended method for this is HTTP Alternative Services . On its first visit to a website, a client connects to a server using TCP. It then discovers via `Alt-Svc` that HTTP/3 is available, and can set up a new QUIC connection. The `Alt-Svc` entry can be cached, allowing subsequent visits to avoid the TCP step, but the entry will eventually become stale and need revalidation. This likely will have to be done for each domain separately, which will probably lead to most page loads using a mix of HTTP/1, HTTP/2, and HTTP/3.

However, even if it is known that a server supports QUIC and HTTP/3, the network in-between might block it. UDP traffic is commonly used in DDoS attacks and blocked by default in for example many company networks. While exceptions could be made for QUIC, its encryption makes it difficult for firewalls to assess the traffic. There are potential solutions to these issues, but in the meantime it is expected that QUIC is most likely to succeed on well-known ports like 443. And it is entirely possible that it is blocked QUIC altogether. In practice, clients will likely use sophisticated mechanisms to fall back to TCP if QUIC fails. One option there is to "race" both a TCP and QUIC connection and use the one that completes first.

There is ongoing work to define ways to discover HTTP/3 without needing the TCP step. This should be considered an optimization though, as the UDP blocking issues are likely to mean that TCP-based HTTP sticks around. The HTTPS DNS record, is similar to HTTP Alternative Services and some CDNs are already experimenting with these records. In the long run, when most servers offer HTTP/3, browsers might switch to attempting that by default. But that will take a long time.

<b>TLS version</b>	<b>HTTP/1 desktop</b>	<b>HTTP/1 mobile</b>	<b>HTTP/2 desktop</b>	<b>HTTP/2 mobile</b>
unknown	4.06%	4.03%	5.05%	7.28%
TLS 1.2	26.56%	24.75%	23.12%	23.14%
TLS 1.3	5.25%	5.11%	35.78%	35.54%

Figure 22.21. TLS adoption by HTTP version.

QUIC is dependent on TLS1.3, which is used for around 41% of requests, as shown in Figure 22.21 which leaves 60% of requests that will need to update their TLS stack to support HTTP/3.

## Is HTTP/3 ready yet?

So, when can we start using HTTP/3 and QUIC for real? Not quite yet, but hopefully soon. There

is a large number of mature open source implementations and the major browsers have experimental support. However, most of the typical servers have suffered some delays: nginx is a bit behind other stacks, Apache hasn't announced official support, and NodeJS relies on OpenSSL, which won't add QUIC support anytime soon. Even so, we expect to see HTTP/3 and QUIC deployments rise throughout 2021.

HTTP/3 and QUIC are highly advanced protocols with powerful performance and security features, such as a new HTTP prioritization system, HOL blocking removal, and 0-RTT connection establishment. This sophistication also makes them difficult to deploy and use correctly, as has turned out to be the case for HTTP/2. We predict that early deployments will mainly be done via the early adoption of CDNs such as Cloudflare, Fastly, and Akamai. This will probably mean that a large part of HTTP/2 traffic can and will be upgraded to HTTP/3 automatically in 2021, giving the new protocol a large traffic share almost out of the box. When and if smaller deployments will follow suit is yet more difficult to answer. Most likely, we will continue to see a healthy mix of HTTP/3, HTTP/2, and even HTTP/1.1 on the web for years to come.

## Conclusion

This year, HTTP/2 has become the dominant protocol, serving 64% of all requests, having grown by 10 percentage points during the year. Home pages have seen a 13% increase in HTTP/2 adoption, leading to an even split of pages served over HTTP/1 and HTTP/2. Using a CDN to serve your home page pushes HTTP/2 adoption up to 80%, compared with 30% for non-CDN usage. There remain some older servers, Apache and IIS, that are proving resistant to upgrading to HTTP/2 and TLS. A big success has been the decrease in website connection usage due to HTTP/2 connection multiplexing. The median number of connections in 2016 was 23 compared to 13 in 2020.

HTTP/2 prioritization and server push have turned out to be way more complex to deploy at large. Under certain implementations they show clear performance benefits. There is, however, a significant barrier to deploying and tuning existing servers to use these features effectively. There are still a large proportion of CDNs who do not support prioritization effectively. There have also been issues with consistent browser support.

HTTP/3 is just around the corner. It will be fascinating to follow the adoption rate, see how discovery mechanisms evolve, and find out which new features will be deployed successfully. We expect next year's Web Almanac to see some interesting new data.

## Authors

---



### Andrew Galloni

@dot\_js dotjs

Andrew works at Cloudflare<sup>42</sup> helping to make the web faster and more secure. He spends his time deploying, measuring and improving new protocols and asset delivery to improve end-user website performance.



### Robin Marx

@programmingart rmarx

Robin is a web protocol and performance researcher at Hasselt University, Belgium<sup>43</sup>. He has been working on getting QUIC and HTTP/3 ready to use by creating tools like qlog and qvis<sup>44</sup>.



### Mike Bishop

MikeBishop

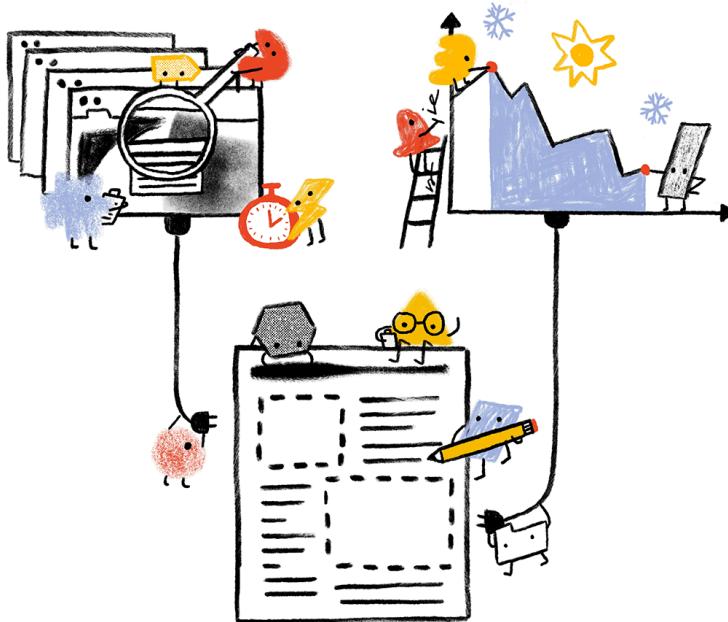
Editor of HTTP/3 with the QUIC Working Group<sup>45</sup>. Architect in Akamai<sup>46</sup>'s Foundry group.

---

42. <https://www.cloudflare.com/>  
43. <https://www.uhasselt.be/edm>  
44. <https://github.com/quiclog>  
45. <https://quicwg.org/>  
46. <https://www.akamai.com/>

# Appendix A

# Methodology



## Overview

The Web Almanac is a project organized by HTTP Archive<sup>47</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>48</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

47. <https://httparchive.org>

48. <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 2020 edition of the Web Almanac is broken into four parts: content, experience, publishing, and distribution. Within each part, several chapters explore their overarching theme from different angles. For example, Part II explores different angles of the user experience in the Performance, Security, and Accessibility chapters, among others.

## About the dataset

The HTTP Archive dataset is continuously updating with new data monthly. For the 2020 edition of the Web Almanac, unless otherwise noted in the chapter, all metrics were sourced from the August 2020 crawl. These results are publicly queryable<sup>49</sup> on BigQuery in tables prefixed with `2020_08_01`.

All of the metrics presented in the Web Almanac are publicly reproducible using the dataset on BigQuery. You can browse the queries used by all chapters in our GitHub repository<sup>50</sup>.

*Please note that some of these queries are quite large and can be expensive<sup>51</sup> to run yourself, as BigQuery is billed by the terabyte. For help controlling your spending, refer to Tim Kadlec's post Using BigQuery Without Breaking the Bank<sup>52</sup>.*

For example, to understand the median number of bytes of JavaScript per desktop and mobile page, see `01_01b.sql`<sup>53</sup>:

```
#standardSQL
# 01_01b: Distribution of JS bytes by client
SELECT
    percentile,
    _TABLE_SUFFIX AS client,
    APPROX_QUANTILES(ROUND(bytesJs / 1024, 2),
    1000) [OFFSET(percentile * 10)] AS js_kbytes
FROM
```

49. [https://github.com/HTTPArchive/httparchive.org/blob/master/docs/gettingstarted\\_bigquery.md](https://github.com/HTTPArchive/httparchive.org/blob/master/docs/gettingstarted_bigquery.md)  
 50. <https://github.com/HTTPArchive/almanac.httparchive.org/tree/main/sql/2020>  
 51. <https://cloud.google.com/bigquery/pricing>  
 52. <https://timkadlec.com/remembers/2019-12-10-using-bigquery-without-breaking-the-bank/>  
 53. [https://github.com/HTTPArchive/almanac.httparchive.org/blob/main/sql/2019/01\\_JavaScript/01\\_01b.sql](https://github.com/HTTPArchive/almanac.httparchive.org/blob/main/sql/2019/01_JavaScript/01_01b.sql)

```

`httparchive.summary_pages.2019_07_01_*`,
UNNEST([10, 25, 50, 75, 90]) AS percentile
GROUP BY
percentile,
client
ORDER BY
percentile,
client

```

Results for each metric are publicly viewable in chapter-specific spreadsheets, for example JavaScript results<sup>54</sup>. Scroll to the bottom of each chapter for links to their queries, results, and comments from readers.

## Websites

There are 7,546,709 websites in the dataset. Among those, 6,347,919 are mobile websites and 5,593,642 are desktop websites. Most websites are included in both the mobile and desktop subsets.

HTTP Archive sources the URLs for its websites from the Chrome UX Report. The Chrome UX Report is a public dataset from Google that aggregates user experiences across millions of websites actively visited by Chrome users. This gives us a list of websites that are up-to-date and a reflection of real-world web usage. The Chrome UX Report dataset includes a form factor dimension, which we use to get all of the websites accessed by desktop or mobile users.

The August 2020 HTTP Archive crawl used by the Web Almanac used the most recently available Chrome UX Report release for its list of websites. The 202006 dataset was released on July 14, 2020 and captures websites visited by Chrome users during the month of June.

There was a 20-30% growth in the number of websites analysed compared to those in the 2019 Web Almanac. This increase has been analysed by Paul Calvano in his Growth of the Web in 2020<sup>55</sup> post.

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

54. [https://docs.google.com/spreadsheets/d/1kBTgIETN\\_V9UjKqK\\_EFmFjRexJnQOmLLr-l2Tkotvic/edit?usp=sharing](https://docs.google.com/spreadsheets/d/1kBTgIETN_V9UjKqK_EFmFjRexJnQOmLLr-l2Tkotvic/edit?usp=sharing)

55. <https://paulcalvano.com/2020-09-29-growth-of-the-web-in-2020/>

the entire website.

HTTP Archive is also considered a lab testing tool, meaning it tests websites from a datacenter and does not collect data from real-world user experiences. All pages are tested with an empty cache in a logged out state, which may not reflect how real users would access them.

## Metrics

HTTP Archive collects thousands of metrics about how the web is built. It includes basic metrics like the number of bytes per page, whether the page was loaded over HTTPS, and individual request and response headers. The majority of these metrics are provided by WebPageTest, which acts as the test runner for each website.

Other testing tools are used to provide more advanced metrics about the page. For example, Lighthouse is used to run audits against the page to analyze its quality in areas like accessibility and SEO. The Tools section below goes into each of these tools in more detail.

To work around some of the inherent limitations of a lab dataset, the Web Almanac also makes use of the Chrome UX Report for metrics on user experiences, especially in the area of web performance.

Some metrics are completely out of reach. For example, we don't necessarily have the ability to detect the tools used to build a website. If a website is built using create-react-app, we could tell that it uses the React framework, but not necessarily that a particular build tool is used. Unless these tools leave detectable fingerprints in the website's code, we're unable to measure their usage.

Other metrics may not necessarily be impossible to measure but are challenging or unreliable. For example, aspects of web design are inherently visual and may be difficult to quantify, like whether a page has an intrusive modal dialog.

## Tools

The Web Almanac is made possible with the help of the following open source tools.

### WebPageTest

WebPageTest<sup>56</sup> is a prominent web performance testing tool and the backbone of HTTP Archive. We use a private instance<sup>57</sup> of WebPageTest with private test agents, which are the actual browsers that test each web page. Desktop and mobile websites are tested under

---

56. <https://www.webpagetest.org/>

57. <https://github.com/WPO-Foundation/webpagetest-docs/blob/master/user/Private%20Instances/README.md>

different configurations:

Config	Desktop	Mobile
Device	Linux VM	Emulated Moto G4
User Agent	Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/ 84.0.4147.105 Safari/537.36 PTST/200805.230825	Mozilla/5.0 (Linux; Android 6.0.1; Moto G (4) Build/MPJ24.139-64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/65.0.3325.146 Mobile Safari/537.36 PTST/200815.130813
Location	Redwood City, California, USA The Dalles, Oregon, USA	Redwood City, California, USA The Dalles, Oregon, USA
Connection	Cable (5/1 Mbps 28ms RTT)	3G (1.600/0.768 Mbps 300ms RTT)
Viewport	1024 x 768px	512 x 360px

Desktop websites are run from within a desktop Chrome environment on a Linux VM. The network speed is equivalent to a cable connection.

Mobile websites are run from within a mobile Chrome environment on an emulated Moto G4 device with a network speed equivalent to a 3G connection. Note that the emulated mobile User Agent self-identifies as Chrome 65 but is actually Chrome 84 under the hood.

There are two locations from which tests are run: California and Oregon USA. HTTP Archive maintains its own test agent hardware located in the Internet Archive<sup>58</sup> datacenter in California. Additional test agents in Google Cloud Platform<sup>59</sup>'s us-west-1 location in Oregon are added as needed.

HTTP Archive's private instance of WebPageTest is kept in sync with the latest public version and augmented with custom metrics<sup>60</sup>. These are snippets of JavaScript that are evaluated on each website at the end of the test. Thanks to the contributions<sup>61</sup> of many data analysts, especially the herculean efforts<sup>62</sup> of Tony McCreath, the 2020 edition of the Web Almanac greatly expanded the capabilities of HTTP Archive's test infrastructure with over 3,000 lines of new code.

The results of each test are made available as a HAR file<sup>63</sup>, a JSON-formatted archive file containing metadata about the web page.

58. <https://archive.org>

59. <https://cloud.google.com/compute/docs/regions-zones/#locations>

60. [https://github.com/HTTPArchive/legacy.httparchive.org/tree/master/custom\\_metrics](https://github.com/HTTPArchive/legacy.httparchive.org/tree/master/custom_metrics)

61. [https://github.com/HTTPArchive/legacy.httparchive.org/commits/master/custom\\_metrics](https://github.com/HTTPArchive/legacy.httparchive.org/commits/master/custom_metrics)

62. <https://github.com/HTTPArchive/legacy.httparchive.org/pulls?q=is%3Apr+author%3ATiggerito+sort%3Acreated-asc>

63. [https://en.wikipedia.org/wiki/HAR\\_\(file\\_format\)](https://en.wikipedia.org/wiki/HAR_(file_format))

## Lighthouse

Lighthouse<sup>64</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 pages are not included because of limited resources. As of the August 2020 crawl, HTTP Archive used the 6.2.0<sup>65</sup> version of Lighthouse.

Lighthouse is run as its own distinct test from within WebPageTest, but it has its own configuration profile:

Config	Value
CPU slowdown	1x/4x
Download throughput	1.6 Mbps
Upload throughput	0.768 Mbps
RTT	150 ms

For more information about Lighthouse and the audits available in HTTP Archive, refer to the Lighthouse developer documentation<sup>66</sup>.

## Wappalyzer

Wappalyzer<sup>67</sup> is a tool for detecting technologies used by web pages. There are 64 categories<sup>68</sup> of technologies tested, ranging from JavaScript frameworks, to CMS platforms, and even cryptocurrency miners. There are over 1,400 supported technologies.

HTTP Archive runs the latest version of Wappalyzer for all web pages. As of August 2020 the Web Almanac used the 6.2.0 version<sup>69</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>70</sup> and CMS<sup>71</sup> categories of technologies detected by Wappalyzer.

---

64. <https://developers.google.com/web/tools/lighthouse/>  
 65. <https://github.com/GoogleChrome/lighthouse/releases/tag/v6.2.0>  
 66. <https://developers.google.com/web/tools/lighthouse/>  
 67. <https://www.wappalyzer.com/>  
 68. <https://www.wappalyzer.com/technologies>  
 69. <https://github.com/AliasIO/Wappalyzer/releases/tag/v6.2.0>  
 70. <https://www.wappalyzer.com/categories/ecommerce>  
 71. <https://www.wappalyzer.com/categories/cms>

All detection tools, including Wappalyzer, have their limitations. The validity of their results will always depend on how accurate their detection mechanisms are. The Web Almanac will add a note in every chapter where Wappalyzer is used but its analysis may not be accurate due to a specific reason.

## Chrome UX Report

The Chrome UX Report<sup>72</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.).

For Web Almanac metrics that reference real-world user experience data from the Chrome UX Report, the August 2020 dataset (202008) is used.

You can learn more about the dataset in the Using the Chrome UX Report on BigQuery<sup>73</sup> guide on web.dev<sup>74</sup>.

## Third Party Web

Third Party Web<sup>75</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>76</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

---

72. <https://developers.google.com/web/tools/chrome-user-experience-report>

73. <https://web.dev/chrome-ux-report-bigquery>

74. <https://web.dev/>

75. <https://www.thirdpartyweb.today/>

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

queried to make the analysis possible. See this thread<sup>77</sup> for more information about how it was integrated with the HTTP Archive dataset on BigQuery.

## Rework Utils

This year's CSS chapter led by Lea Verou took a significantly more detailed look at the state of CSS, spread over 100+ queries<sup>78</sup>. For perspective, that's 2.5x more queries than in 2019. To make this scale of analysis feasible, Lea open sourced the Rework Utils<sup>79</sup>. These utilities take the JSON data from Rework to the next level by providing helpful scripts to more easily extract CSS insights. Most of the stats you see in the CSS chapter are powered by these scripts.

## Parsel

Parsel<sup>80</sup> is a CSS selector parser and specificity calculator, originally written by CSS chapter lead Lea Verou and open sourced as a separate library. It is used extensively in all CSS metrics that relate to selectors and specificity.

## Analytical process

The Web Almanac took about a year to plan and execute with the coordination of more than a hundred contributors from the web community. This section describes why we chose the chapters you see in the Web Almanac, how their metrics were queried, and how they were interpreted.

## Planning

The 2020 Web Almanac kicked off in June 2020<sup>81</sup>, later than the 2019 timeline due to the unrest related to COVID-19 and the social justice protests. These and other events of 2020 were an undercurrent throughout the entire production process and put a lot of additional strain on our contributors beyond the stresses of a fast-paced project like this.

As we stated in last year's Methodology<sup>82</sup>:

*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.*

---

77. <https://discuss.httparchive.org/t/analyzing-style-sheets-with-a-js-based-parser/1683>

78. [https://github.com/HTTPArchive/almanac.httparchive.org/tree/main/sql/2020/01\\_CSS](https://github.com/HTTPArchive/almanac.httparchive.org/tree/main/sql/2020/01_CSS)

79. <https://github.com/LeaVerou/rework-utils>

80. <https://projects.verou.me/parsel/>

81. [https://twitter.com/rick\\_visconi/status/1273135952848977920](https://twitter.com/rick_visconi/status/1273135952848977920)

82. <https://almanac.httparchive.org/en/2019/methodology#brainstorming>

To that end, this year we've made systematic changes to the way that we seek and select authors:

- Previous authors were specifically discouraged from writing again to make room for different perspectives.
- Everyone endorsing 2020 authors were asked to be especially conscious not to nominate people who all look or think alike.
- Many 2019 authors were Google employees and this year we tried to get a greater balance of perspectives from the broader web community. We believe that the voices in the Web Almanac should be a reflection of the community itself, and not skewed towards any specific company to avoid creating echo chambers.
- The project leads reviewed all of the author nominations and made an effort to select authors who will bring new perspectives and amplify the voices of underrepresented groups in the community.

We hope to iterate on this process in the future to ensure that the Web Almanac is a more diverse and inclusive project with contributors from all backgrounds.

Finally, in July 2020, after rounds of brainstorming and nominations, 22 chapters were solidified and we formed content teams for each chapter tasked with writing, reviewing, and analysis.

## **Analysis**

In July and August 2020, with the stable list of metrics and chapters, data analysts triaged the metrics for feasibility. In some cases, custom metrics<sup>83</sup> were created to fill gaps in our analytic capabilities.

Throughout August 2020, the HTTP Archive data pipeline crawled several million websites, gathering the metadata to be used in the Web Almanac.

The data analysts began writing queries to extract the results for each metric. In total, hundreds of queries were written by hand! You can browse all of the queries by year and chapter in our open source query repository<sup>84</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.

---

83. [https://github.com/HTTPArchive/legacyhttparchive.org/tree/master/custom\\_metrics](https://github.com/HTTPArchive/legacyhttparchive.org/tree/master/custom_metrics)

84. <https://github.com/HTTPArchive/almanac.httparchive.org/tree/main/sql/2020>

To make the results more easily understandable to readers, web developers and analysts created data visualizations to embed in the chapter. Some visualizations are simplified to make the points more clearly. For example, rather than showing a full distribution, only a handful of percentiles are shown. Unless otherwise noted, all distributions are summarized using percentiles, especially medians (the 50th percentile), and not averages.

Finally, editors revised the chapters to fix simple grammatical errors and ensure consistency across the reading experience.

## Looking ahead

The 2020 edition of the Web Almanac is the second in what we hope to continue as an annual tradition in the web community of introspection and a commitment to positive change. Getting to this point has been a monumental effort thanks to many dedicated contributors and we hope to leverage as much of this work as possible to make future editions even more streamlined.

If you're interested in contributing to the 2021 edition of the Web Almanac, please fill out our interest form<sup>85</sup>. Let's work together to track the state of the web!

---

85. <https://forms.gle/VRBFegGAP7d99Bhp7>

# Appendix B

# Contributors



The Web Almanac has been made possible by the hard work of the web community. 119 people have volunteered countless hours in the planning, research, writing and production phases.



Abby Tsai

[@AbbyTsai](#)

Analysts, Developers, Translators



Alex Denning

[@AlexDenning](#)

[@alexdenning](#)

<https://getellipsis.com>

Authors



Aditya Pandey

[@adityapandey98](#)

[@adityapandey1998](#)

[adityapandey98](#)

Developers



Alex Tait

[@at\\_fresh\\_dev](#)

[@alextait1](#)

<https://togetherthetech.ca/>

Authors



Adrian Roselli

[@aardrian](#)

[@aardrian](#)

<https://adrianroselli.com/>

Reviewers



Alexey Pyltsyn

[@lex111](#)

<https://lex111.ru/>

Translators



Ahmad Awais

[@MrAhmadAwais](#)

[@ahmadawais](#)

<https://AhmadAwais.com>

Authors Reviewers



Aleyda Solis

[@aleyda](#)

[@aleyda](#)

<http://www.aleydasolis.com/en/>

Authors



Akshar Agarwal

[@AAgar](#)

[@AAgar](#)

<https://aagar.github.io/>

Analysts



Andrew Galloni

[@dot\\_js](#)

[@dotjs](#)

Authors



Alberto Medina

[@iAlbMedina](#)

[@amedina](#)

Reviewers



Andy Bell

[@hankchizljaw](#)

[@hankchizljaw](#)

<https://hankchizljaw.com/>

Reviewers

	<b>Antoine Eripert</b> ⌚ antoineeripert Analysts		<b>Caleb Queern</b> ⌚ @cqueern ⌚ cqueern Reviewers
	<b>Artem Denysov</b> 🐦 @denar90_ ⌚ denar90 Analysts		<b>Catalin Rosu</b> 🐦 @catalinred ⌚ catalinred 🌐 https://catalin.red/ Authors, Developers, Reviewers
	<b>Barry Pollard</b> 🐦 @tunetheweb ⌚ bazzadp 💻 barry-pollard-developer 🌐 https://www.tunetheweb.com Analysts, Developers, Editors, Project Leads, Reviewers		<b>Cheney Tsai</b> ⌚ cheneysai Reviewers
	<b>Ben Seymour</b> ⌚ bseymour 🌐 http://benseymour.com Authors		<b>Cheng Xi</b> ⌚ chengxiin Translators
	<b>Bharat Agarwal</b> ⌚ bharatagsrwal 🌐 https://iambharat.me Developers		<b>Chris Lilley</b> 🐦 @svgeesus ⌚ svgeesus 🌐 http://svgees.us/ Authors Reviewers
	<b>Boris Schapira</b> 🐦 @boostmarks ⌚ borisschapira 🌐 https://boris.schapira.dev Developers, Reviewers, Translators		<b>Christian Liebel</b> 🐦 @christianliebel ⌚ christianliebel 🌐 https://christianliebel.com Authors
	<b>Brian Kardell</b> 🐦 @briankardell ⌚ bkardell 🌐 https://bkardell.com Reviewers		<b>Colin Bendell</b> 🐦 @colinbendell ⌚ colinbendell Reviewers
	<b>Brian Rinaldi</b> 🐦 @remotesynth ⌚ remotesynth 🌐 https://remotesynthesis.com/ Analysts		<b>Dave Crossland</b> 🐦 @davelab6 ⌚ davelab6 🌐 http://fonts.google.com Reviewers
			<b>Dave Smart</b> 🐦 @davewsmart ⌚ dwsmart 🌐 https://tamethebots.com/ Reviewers

**Dave Sottimano**

✉ @dsottimano  
👤 dsottimano  
🌐 https://opensourceseo.org/  
Reviewers

**David Fox**

✉ @theoboto  
👤 oboto  
🌐 https://www.lookzook.com  
Analysts, Editors, Project Leads,  
Reviewers

**Doug Sillars**

✉ @dougsillars  
👤 dougsillars  
🌐 https://dougsillars.com  
Reviewers

**Drewz**

👤 drewzboto  
Reviewers

**Durga Prasad Sadhanala**

✉ @dsadhanala  
👤 dsadhanala  
🌐 https://web-ui.dev  
Developers

**Dustin Montgomery**

✉ @DustinMontSEO  
👤 en3r0  
🌐 https://dustinmontgomery.com/  
Reviewers

**Edmond W. W. Chan**

👤 edmondwwchan  
🌐 https://edmondwwchan.github.io/  
Reviewers

**Eric Bailey**

✉ @ericwbailey  
👤 ericwbailey  
🌐 https://ericwbailey.design/  
Reviewers

**Fantasai**

✉ @fantasai  
👤 fantasai  
🌐 http://fantasai.inkedblade.net/  
Reviewers

**Giovanni Punti**

✉ @giopunt  
👤 giopunt  
Reviewers

**Gokulakrishnan Kalaikovan**

👤 gokulkrishh  
Reviewers

**Greg Brimble**

✉ @GregBrimble  
👤 GregBrimble  
🌐 https://gregbrimble.com/  
Analysts

**Greg Wolf**

👤 gregorywolf  
Analysts

**Harry Roberts**

✉ @csswizardry  
👤 csswizardry  
🌐 https://csswizardry.com/  
Reviewers

**Henri Helvetica**

✉ @HenriHelvetica  
👤 henrihelvetica  
Authors

**Ian Devlin**

✉ @iandevlin  
👤 iandevlin  
🌐 https://iandevlin.com  
Authors

**Jad Joubran**

✉ @JoubranJad  
👤 jadoubran  
🌐 https://learnjavascript.online/  
Reviewers



**Jai Santhosh**

Twitter: @jaisanth  
GitHub: jaisanth  
Website: <https://jaisanth.com/>  
Role: Reviewers



**Jamie Indigo**

Twitter: @Jammer\_Volts  
GitHub: fellowhuman1101  
Website: <https://not-a-robot.com/>  
Role: Authors



**Jason Haralson**

GitHub: jrharalson  
Role: Reviewers



**Jason Pamental**

Twitter: @jpamental  
GitHub: jpamental  
Website: <http://rwt.io>  
Role: Authors



**Jens Oliver Meiert**

Twitter: @j9t  
GitHub: j9t  
Website: <https://meiert.com/en/>  
Role: Authors Reviewers



**Jeremy Wagner**

Twitter: @malchata  
GitHub: malchata  
Website: <https://jeremy.codes/>  
Role: Reviewers



**Jessica Nicolet**

Twitter: @jessica\_nicolet  
GitHub: Jessnicolet  
Website: <https://www.jessicanicolet.com/>  
Role: Reviewers



**Jonathan Wold**

Twitter: @sirjonathan  
GitHub: sirjonathan  
Website: <https://jonathanwold.com>  
Role: Reviewers



**Jyrki Alakuijala**

GitHub: jyrkialakuijala  
Role: Authors



**Karolina Szczur**

Twitter: @fox  
GitHub: thefoxis  
Role: Authors



**Katie Hempenius**

Twitter: @katiehempenius  
GitHub: khempenius  
Role: Analysts Reviewers



**Laurent Devernay**

Twitter: @ldevernay  
GitHub: ldevernay  
Website: <https://ldevernay.github.io/>  
Role: Reviewers



**Lea Verou**

Twitter: @leaverou  
GitHub: LeaVerou  
Website: <http://lea.verou.me/>  
Role: Authors



**Leonardo Zizzamia**

Twitter: @Zizzamia  
GitHub: Zizzamia  
Website: <https://twitter.com/zizzamia>  
Role: Authors Reviewers



**Luca Versari**

GitHub: veluca93  
Role: Authors



**Lucas Pardue**

Twitter: @SimmerVigor  
GitHub: LPardue  
Website: <https://lucaspardue.com>  
Role: Reviewers



**Lyubomir Angelov**

Twitter: @angelovcode  
GitHub: Super-Fly  
Role: Developers



**Maedah Batool**

Twitter: @maedahbatool  
GitHub: MaedahBatool  
Website: <https://maedahbatool.com/>  
Role: Reviewers

**Mandy Michael**

✉ @Mandy\_Kerr  
 🌐 mandymichael  
 Ⓛ <https://mandymichael.com/>  
 Reviewers

**Miriam Suzanne**

✉ @MiriSuzanne  
 🌐 mirisuzanne  
 Ⓛ <http://miriamsuzanne.com/>  
 Reviewers

**Manuel Matuzovic**

✉ @mmatuzzo  
 🌐 matuzo  
 Ⓛ <https://www.matuzo.at/>  
 Reviewers

**Moritz Firsching**

✉ mo271  
 Ⓛ <https://mo271.github.io/>  
 Authors

**Max Ostapenko**

✉ @themax\_o  
 🌐 max-ostapenko  
 Ⓛ <https://maxostapenko.com>  
 Analysts Developers

**Nate Dame**

✉ @seonate  
 🌐 natedame  
 Reviewers

**Michael DiBlasio**

✉ mdiblasio  
 🌐 Authors

**Nicolas Goutay**

✉ @Phacks  
 🌐 phacks  
 Ⓛ <https://phacks.dev/>  
 Reviewers

**Michael King**

✉ @IPullRank  
 🌐 ipullrank  
 Authors

**Nicolas Hoizey**

✉ @nhoizey  
 🌐 nhoizey  
 Ⓛ <https://nicolas-hoizey.com/>  
 Reviewers

**Michelle O'Connor**

Designers

**Noam Rosenthal**

✉ @nomsternom  
 🌐 noamr  
 Reviewers

**Miguel Carlos Martínez Díaz**

✉ @mcmd  
 🌐 mcmd  
 Ⓛ [miguelcarlosmartinezdiaz](https://miguelcarlosmartinezdiaz)  
 Translators

**Nurullah Demir**

✉ @nrllah  
 🌐 nrllh  
 Ⓛ <https://internet-sicherheit.de>  
 Authors

**Mike Bishop**

✉ MikeBishop  
 🌐 Authors

**Olu Niyi-Awosusi**

✉ oluoluoxenfree  
 Ⓛ <http://www.opentagclosetag.com/>  
 Authors

**Minko Gechev**

✉ @mgechev  
 🌐 mgechev  
 Ⓛ <https://blog.mgechev.com/>  
 Reviewers

**Pascal Schilp**

✉ thepassle  
 Reviewers

	<b>Patrick Meenan</b> Twitter: @patmeenan GitHub: pmeenan Website: <a href="https://www.webpagetest.org/">https://www.webpagetest.org/</a> Role: Reviewers		<b>Renee Johnson</b> Twitter: @reneesoffice GitHub: ernee Website: <a href="https://reneesvirtualoffice.com">https://reneesvirtualoffice.com</a> Role: Reviewers
	<b>Paul Calvano</b> Twitter: @paulcalvano GitHub: paulcalvano Website: <a href="https://paulcalvano.com">https://paulcalvano.com</a> Role: Analysts, Developers, Project Leads, Reviewers		<b>Rick Viscomi</b> Twitter: @rick_viscomi GitHub: rviscomi Role: Analysts, Developers, Editors, Project Leads, Reviewers
	<b>Pearl Latteier</b> Twitter: @pblatteier GitHub: pearlbea Role: Reviewers		<b>Robin Marx</b> Twitter: @programmingart GitHub: rmarx Role: Authors
	<b>Pokidov N. Dmitry</b> Twitter: @otherpunk GitHub: dooman87 Website: <a href="https://pixboost.com">https://pixboost.com</a> Role: Analysts		<b>Rockey Nebhwani</b> Twitter: @rnebhwanı GitHub: rockynebhwanı Website: <a href="https://rockynebhwanı.com">rockynebhwanı.com</a> Role: Authors
	<b>Praveen Pal</b> Twitter: @PraveenPal4232 GitHub: PraveenPal4232 Website: <a href="https://praveenpal4232.github.io">https://praveenpal4232.github.io</a> Role: Translators		<b>Roel Nieskens</b> Twitter: @PixelAmbacht GitHub: RoelN Website: <a href="http://pixelambacht.nl/">http://pixelambacht.nl/</a> Role: Reviewers
	<b>Rachel Andrew</b> Twitter: @rachelandrew GitHub: rachelandrew Website: <a href="https://rachelandrew.co.uk/">https://rachelandrew.co.uk/</a> Role: Authors		<b>Rory Hewitt</b> GitHub: roryhewitt Role: Authors
	<b>Raghu Ramakrishnan</b> GitHub: raghuramakrishnan71 Role: Analysts Authors		<b>Sakae Kotaro</b> Twitter: @beltway7 GitHub: ksakae1216 Website: <a href="https://www.ksakae1216.com/archive">https://www.ksakae1216.com/archive</a> Role: Translators
	<b>Raph Levien</b> Twitter: @raphlinus GitHub: raphlinus Website: <a href="https://levien.com">https://levien.com</a> Role: Authors		<b>Sami Boukortt</b> GitHub: sboukortt Role: Authors
	<b>Saptak Sengupta</b> Twitter: @Saptak013 GitHub: saptaks Website: <a href="https://saptaks.website/">https://saptaks.website/</a> Role: Developers		

**Sawood Alam**

✉ @ibnesayeed  
 🌐 ibnesayeed  
 🌐 <https://www.cs.odu.edu/~salam/>  
 Developers Reviewers

**Tammy Everts**

✉ @tameverts  
 🌐 tammyeverts  
 🌐 <https://speedcurve.com/>  
 Reviewers

**Shane Exterkamp**

✉ @Shane\_Exterkamp  
 🌐 exterkamp  
 Editors Reviewers

**Thomas Steiner**

✉ tomayac  
 🌐 <https://blog.tomayac.com/>  
 Analysts Reviewers

**Shubhie Panicker**

✉ @shubhie  
 🌐 spanicker  
 Authors

**Tim Kadlec**

✉ @tkadlec  
 🌐 tkadlec  
 Authors

**Simon Hearne**

✉ @simonhearne  
 🌐 simonhearne  
 🌐 <https://simonhearne.com>  
 Authors

**Tom Van Goethem**

✉ @tomvangoethem  
 🌐 tomvangoethem  
 Analysts Authors

**Simon Pieters**

✉ @zcorpan  
 🌐 zcorpan  
 Reviewers

**Tony McCreath**

✉ @TonyMcCreath  
 🌐 Tiggerito  
 🌐 <https://websiteadvantage.com.au/>  
 Analysts

**Soham-S-Sarkar**

✉ Soham-S-Sarkar  
 Reviewers

**William Sandres**

✉ @hakacode  
 🌐 HakaCode  
 🌐 <https://hakacode.github.io>  
 Translators

**Stefan Matei**

✉ @smatei  
 🌐 smatei  
 🌐 <https://www.advancedwebranking.com/>  
 Analysts

**Yana Dimova**

✉ ydimova  
 Analysts Authors

**Sudheendra chari**

✉ @itsmesudheendra  
 🌐 sudheendrachari  
 🌐 [sudheendrachari](https://sudheendrachari.com)  
 Developers

**hemanth.hm**

✉ @gnumanth  
 🌐 hemanth  
 🌐 <http://h3manth.com>  
 Authors

**Tamas Piros**

✉ @tpiros  
 🌐 tpiros  
 🌐 <https://www.fullstacktraining.com>  
 Authors

**jzyang**

✉ jzyang  
 Reviewers



**notwillk**  
notwillk  
Reviewers



**rsheeter**  
rsheeter  
Reviewers