Loading Third-Party JavaScript

By Addy Osmani
Eng Manager, Web Developer Relations

By Arthur Evans
Arthur is a Tech Writer

You've optimized all of your code, but your site still loads too slowly. Who's the culprit?

Often, performance problems slowing pages down are due to third-party scripts: ads, analytics, trackers, social-media buttons, and so on.

Third-party scripts provide a wide range of useful functionality, making the web more dynamic, interactive, and interconnected. These scripts may be crucial to your website's functionality or revenue stream. But third-party scripts also come with **many risks** that should be taken into consideration to **minimize their impact** while still providing value.

Why do you need to be careful about third-party scripts?

- They can be a **performance** concern
- They can be a **privacy** concern
- They might be a **security** concern
- They can be **unpredictable** and change without you knowing
- They can have **unintended consequences**

Ideally, you'll want to ensure third-party script is not impacting the critical rendering path. In this guide, we'll walk through how to find and fix issues related to loading third-party JavaScript.

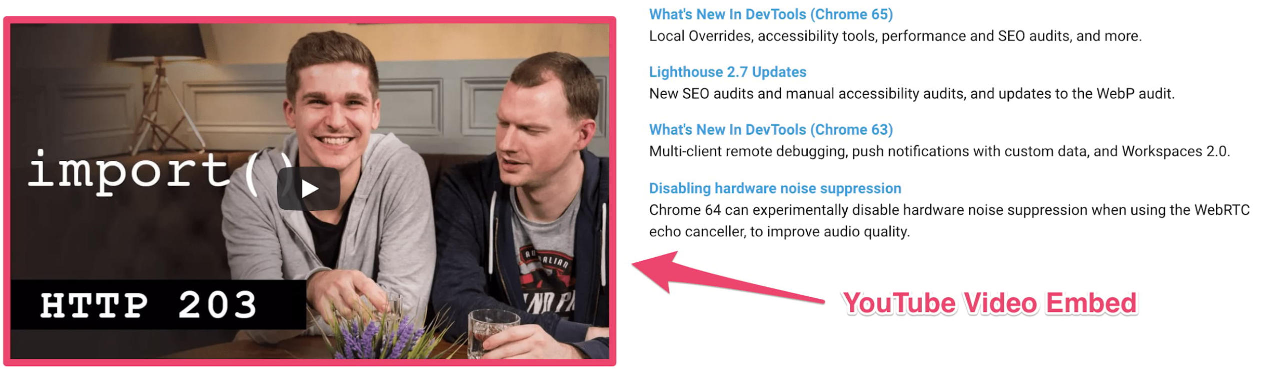
What do we mean by third-party scripts?

Third-party JavaScript often refers to scripts that can be embedded into any site directly from a third-party vendor. These scripts can include ads, analytics, widgets and other scripts that make the web more dynamic and interactive.

Examples of third-party scripts include:

- Social sharing buttons (e.g Twitter, Facebook, G+)
- Video player embeds (e.g YouTube, Vimeo)

- Advertising iframes
- Analytics & metrics scripts
- A/B testing scripts for experiments
- Helper libraries (e.g date formatting, animation, functional libraries etc)



What's New In DevTools (Chrome 65)
Local Overrides, accessibility tools, performance and SEO audits, and more.

Lighthouse 2.7 Updates
New SEO audits and manual accessibility audits, and updates to the WebP audit.

What's New In DevTools (Chrome 63)
Multi-client remote debugging, push notifications with custom data, and Workspaces 2.0.

Disabling hardware noise suppression
Chrome 64 can experimentally disable hardware noise suppression when using the WebRTC echo canceller, to improve audio quality.

YouTube Video Embed

```
<iframe
  width="560" height="315" src="https://www.youtube.com/embed/mo8thg5XGV0"
  frameborder="0" allow="autoplay; encrypted-media" allowfullscreen>
</iframe>
```

One example of this is the YouTube video player embed script that allows you to embed a video into your page.

Unfortunately, embedding third-party scripts means we often rely on them to be fast in order to avoid slowing our pages down. Third-party scripts are a predominant cause of performance slowdowns and are often caused by resources outside of your control.

These issues can include:

- Firing too many network requests to multiple servers. The more requests a site has to make, the longer it can take to load.
- Sending too much JavaScript that keeps the main thread busy. Too much JavaScript can block DOM construction, delaying how quickly pages can render. CPU-intensive script parsing and execution can delay user interaction and cause battery drain.
- Sending large, unoptimized image files or videos. This can consume data and cost users money.
- Third-party scripts loaded without care can be a single-point of failure (SPOF)
- Insufficient HTTP caching, forcing resources to be fetched from the network often
- Lack of sufficient server compression of resources

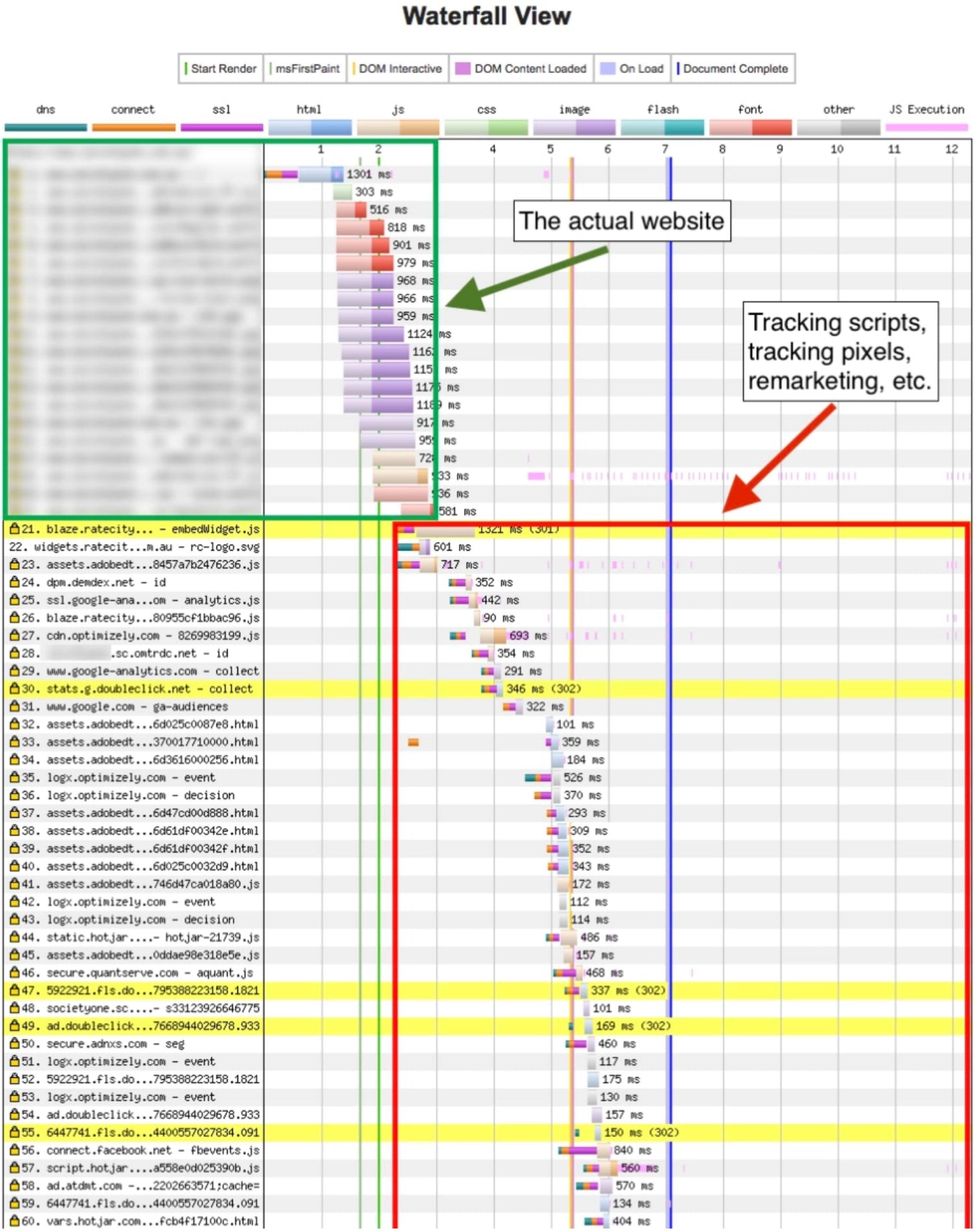
- Blocking content display until they complete processing. This can also be true for async A/B testing scripts.
- Use of legacy APIs (e.g document.write()) known to be harmful to the user experience
- Excessive DOM elements or expensive CSS selectors.
- Including multiple third party embeds can lead to multiple frameworks and libraries being pulled in several times. This is wasteful and exacerbates the performance issues.
- Third-party scripts often use embed techniques that can block window.onload if their servers respond slowly, even if the embed is using async or defer.

Context is important and the solution to costly third-parties can depend on your site and ability to configure how third-party code is being loaded. Thankfully a number of solutions and tools exist to find and fix issues with third party resources.

How do you identify third-party script on a page?

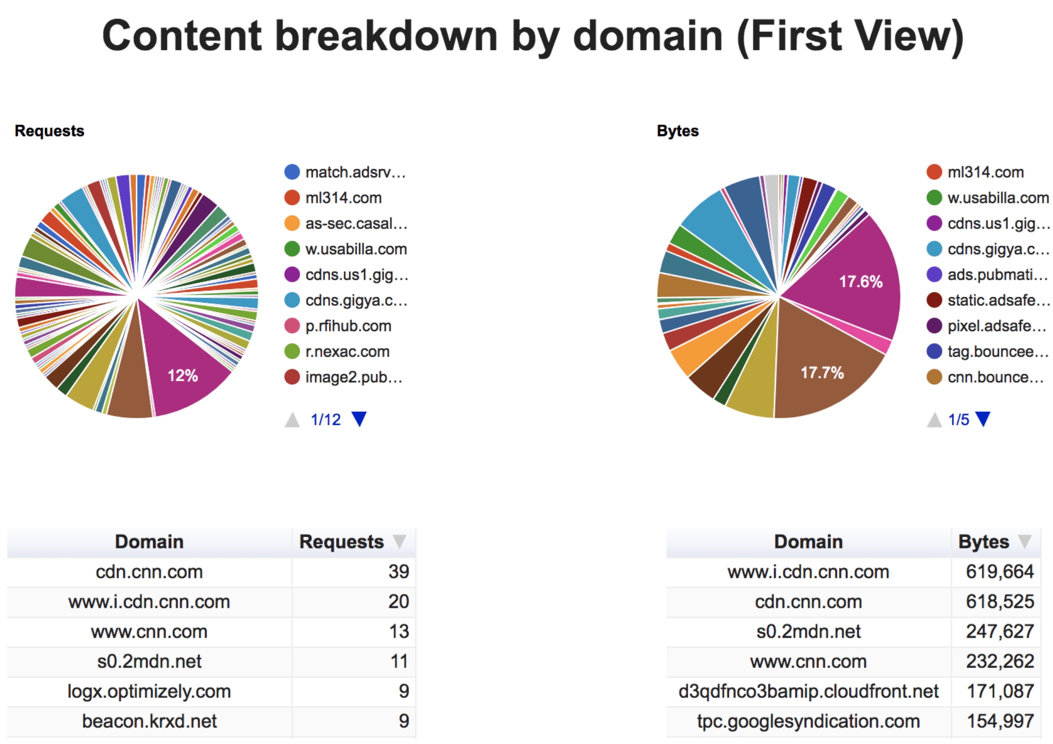
Unless you're aware which third-party scripts are loaded by your site and what their performance impact is, it's impossible to know how to optimize them. Many free web speed test tools can highlight costly third-parties including Chrome DevTools, PageSpeed Insights and WebPageTest. These tools display rich diagnostic information that can tell you *how many* third party scripts are loaded by your site and which take the most time to execute.

WebPageTest's waterfall view can highlight the impact of heavy third-party script use. Below is an example of the requests required to load the main content for a site vs. the tracking and marketing scripts (credit: Tags Gone Wild).



21. blaze.ratecity... - embedWidget.js 1321 ms (301)
22. widgets.ratecit...m.au - rc-logo.svg 601 ms
23. assets.adobedt...8457a7b2476236.js 717 ms
24. dpm.demdex.net - id 352 ms
25. ssl.google-ana...om - analytics.js 442 ms
26. blaze.ratecity...80955cf1bbac96.js 90 ms
27. cdn.optimizely.com - 8269983199.js 693 ms
28. [redacted].sc.omtrdc.net - id 354 ms
29. www.google-analytics.com - collect 291 ms
30. stats.g.doubleclick.net - collect 346 ms (302)
31. www.google.com - ga-audiences 322 ms
32. assets.adobedt...6d025c0087e8.html 101 ms
33. assets.adobedt...370017710000.html 359 ms
34. assets.adobedt...6d3616000256.html 184 ms
35. logx.optimizely.com - event 526 ms
36. logx.optimizely.com - decision 370 ms
37. assets.adobedt...6d47cd00d888.html 293 ms
38. assets.adobedt...6d61df00342e.html 309 ms
39. assets.adobedt...6d61df00342f.html 352 ms
40. assets.adobedt...6d025c0032d9.html 343 ms
41. assets.adobedt...746d47ca018a80.js 172 ms
42. logx.optimizely.com - event 112 ms
43. logx.optimizely.com - decision 114 ms
44. static.hotjar....- hotjar-21739.js 486 ms
45. assets.adobedt...0ddae98e318e5e.js 157 ms
46. secure.quantserve.com - aquant.js 468 ms
47. 5922921.fls.do...795388223158.1821 337 ms (302)
48. societyone.sc....- s33123926646775 101 ms
49. ad.doubleclick...7668944029678.933 169 ms (302)
50. secure.adnxs.com - seg 460 ms
51. logx.optimizely.com - event 117 ms
52. 5922921.fls.do...795388223158.1821 175 ms
53. logx.optimizely.com - event 130 ms
54. ad.doubleclick...7668944029678.933 157 ms
55. 6447741.fls.do...4400557027834.091 150 ms (302)
56. connect.facebook.net - fbevents.js 840 ms
57. script.hotjar....a558e0d025390b.js 560 ms
58. ad.atdmt.com -...2202663571;cache= 570 ms
59. 6447741.fls.do...4400557027834.091 134 ms
60. vars.hotjar.com...fcb4f17100c.html 404 ms

WebPageTest’s domain breakdown can also be useful for visualizing how much content comes from third-party origins. It breaks this down by both total bytes and the number of requests:



Content breakdown by domain (First View)

Domain	Requests ▼
cdn.cnn.com	39
www.i.cdn.cnn.com	20
www.cnn.com	13
s0.2mdn.net	11
logx.optimizely.com	9
beacon.krxd.net	9

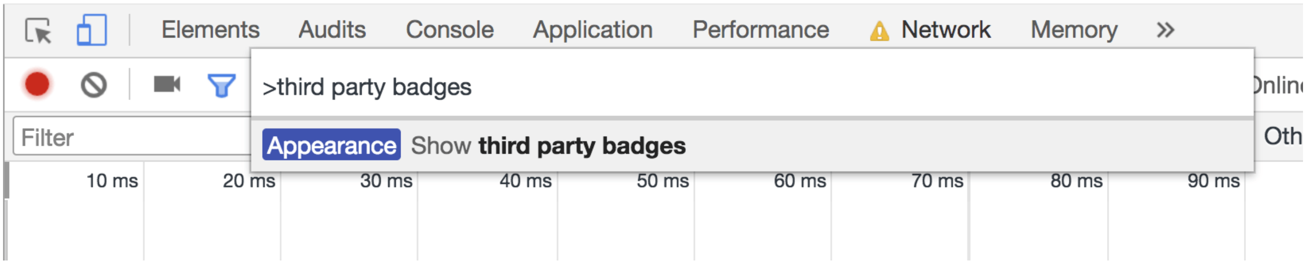
Domain	Bytes ▼
www.i.cdn.cnn.com	619,664
cdn.cnn.com	618,525
s0.2mdn.net	247,627
www.cnn.com	232,262
d3qdfnco3bamip.cloudfront.net	171,087
tpc.googlesyndication.com	154,997

When you see a problematic script, figure out what the script does and ask yourself whether the script is really that necessary. Do an A/B test to balance the perceived value versus its impact on key user engagement or performance metrics.

Chrome DevTools Third-party Script Badging

Chrome DevTools has support for highlighting third-parties (by their product name) in the Network panel. This allows you to get more insight into the third-parties making request on a page, logging to the console and executing expensive JavaScript on your page.

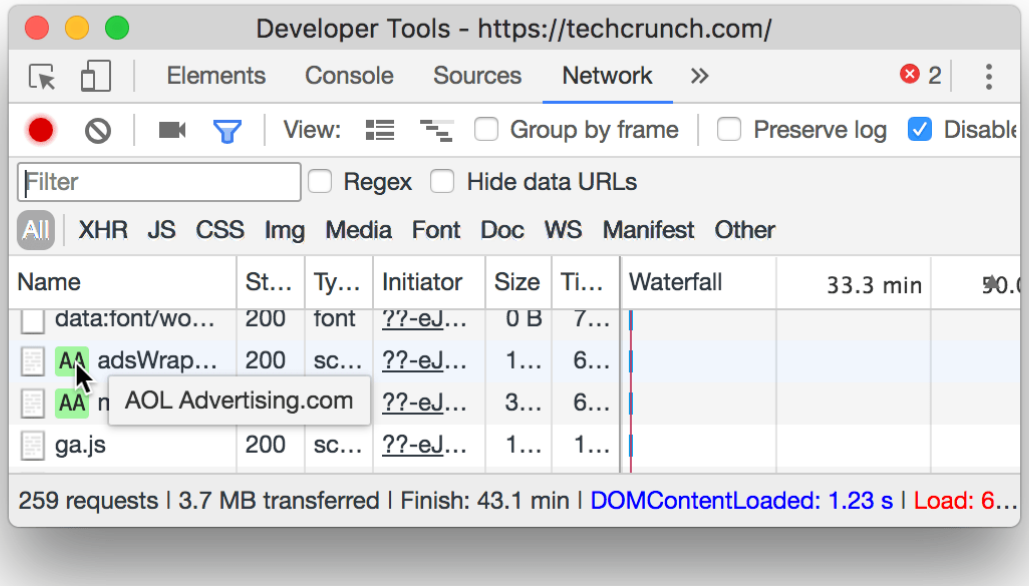
To show third party badges, navigate to any panel in the Chrome DevTools and hit CMD + Shift + P to bring up the Command Menu. Next enter in "Show third party badges". This will enable the feature:



>third party badges

Appearance Show **third party badges**

When you record a page load using the Network panel it will now include third party badges, like the "AOL Advertising" badges shown below in green. Hovering over a third-party badge in the Network panel will display more information about that script, helping you identify what it does.



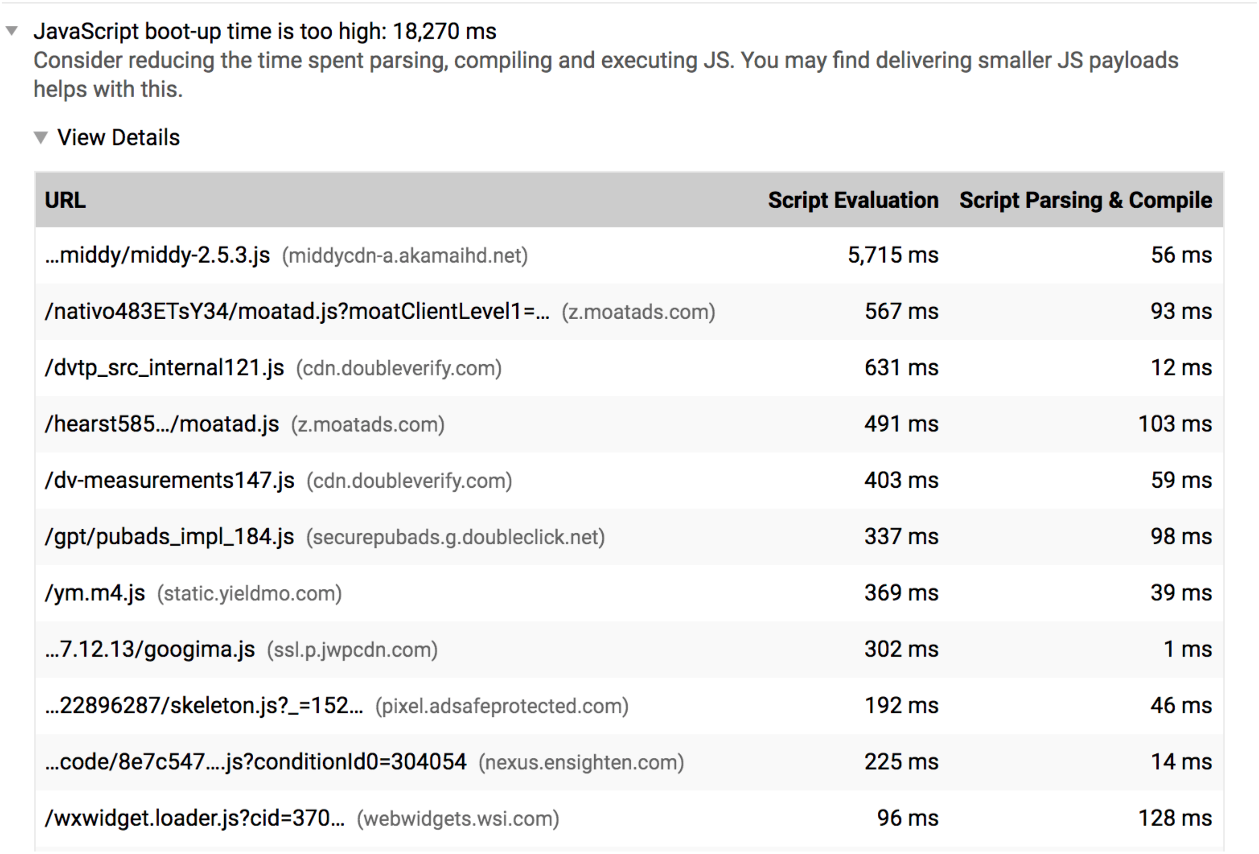
Name	St…	Ty…	Initiator	Size	Ti…	Waterfall	33.3 min	50.0
data:font/wo…	200	font	??-eJ…	0 B	7…			
AA adsWrap…	200	sc…	??-eJ…	1…	6…			
AA n			??-eJ…	3…	6…			
ga.js	200	sc…	??-eJ…	1…	1…			

259 requests | 3.7 MB transferred | Finish: 43.1 min | DOMContentLoaded: 1.23 s | Load: 6…

How do I measure the impact of third-party script on my page?

Lighthouse Boot-up Time Audit

The Lighthouse JavaScript boot-up time audit highlights scripts that have a costly script parse, compile or evaluation time. This can be useful for discovering CPU-intensive third party scripts.



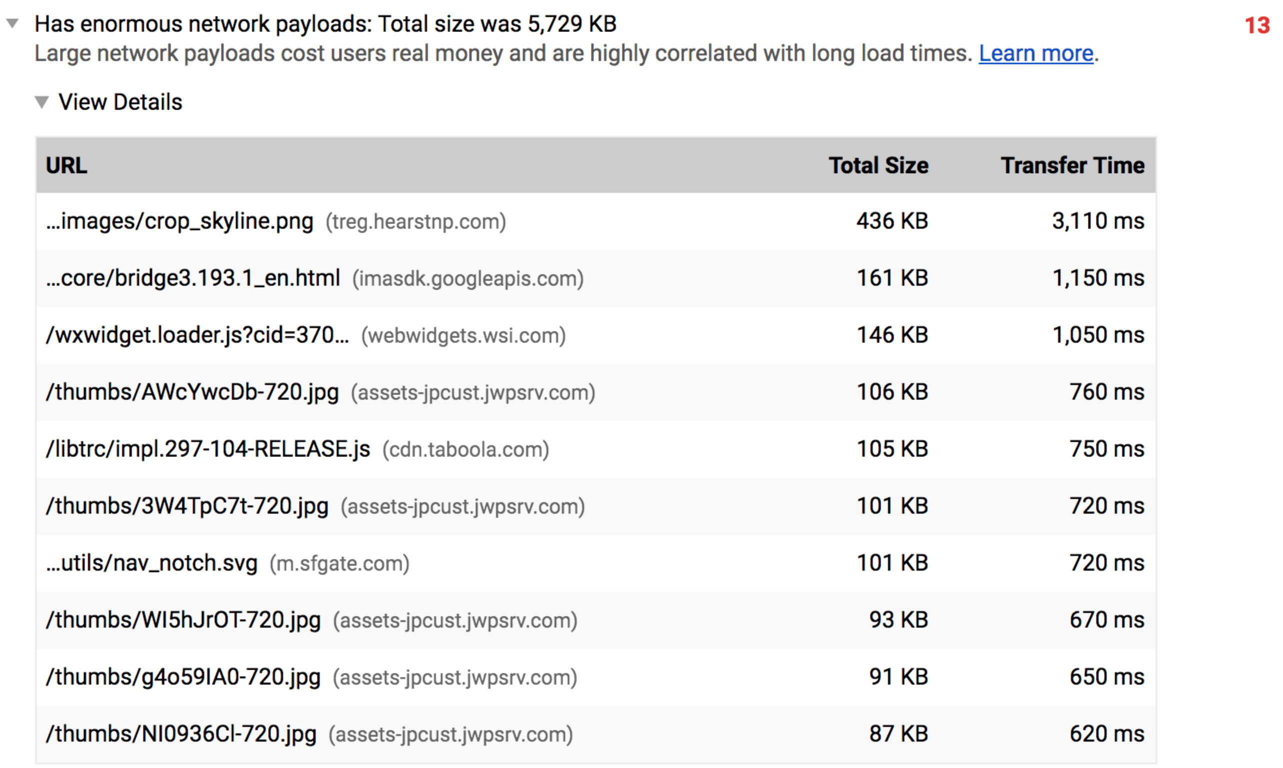
▼ JavaScript boot-up time is too high: 18,270 ms
Consider reducing the time spent parsing, compiling and executing JS. You may find delivering smaller JS payloads helps with this.

▼ View Details

URL	Script Evaluation	Script Parsing & Compile
…middy/middy-2.5.3.js (middycdn-a.akamaihd.net)	5,715 ms	56 ms
/nativo483ETsY34/moatad.js?moatClientLevel1=… (z.moatads.com)	567 ms	93 ms
/dvtp_src_internal121.js (cdn.doubleverify.com)	631 ms	12 ms
/hearst585…/moatad.js (z.moatads.com)	491 ms	103 ms
/dv-measurements147.js (cdn.doubleverify.com)	403 ms	59 ms
/gpt/pubads_impl_184.js (securepubads.g.doubleclick.net)	337 ms	98 ms
/ym.m4.js (static.yieldmo.com)	369 ms	39 ms
…7.12.13/googima.js (ssl.p.jwpcdn.com)	302 ms	1 ms
…22896287/skeleton.js?_=152… (pixel.adsafeprotected.com)	192 ms	46 ms
…code/8e7c547….js?conditionId0=304054 (nexus.ensighten.com)	225 ms	14 ms
/wxwidget.loader.js?cid=370… (webwidgets.wsi.com)	96 ms	128 ms

Lighthouse Network Payloads Audit

The Lighthouse Network Payloads audit identifies network requests (including those from third-parties) that may slow down page load time. Avoiding these requests or highlighting their cost to ad-networks can save users money they would have spent on cellular data.



Has enormous network payloads: Total size was 5,729 KB

13

Large network payloads cost users real money and are highly correlated with long load times. Learn more.

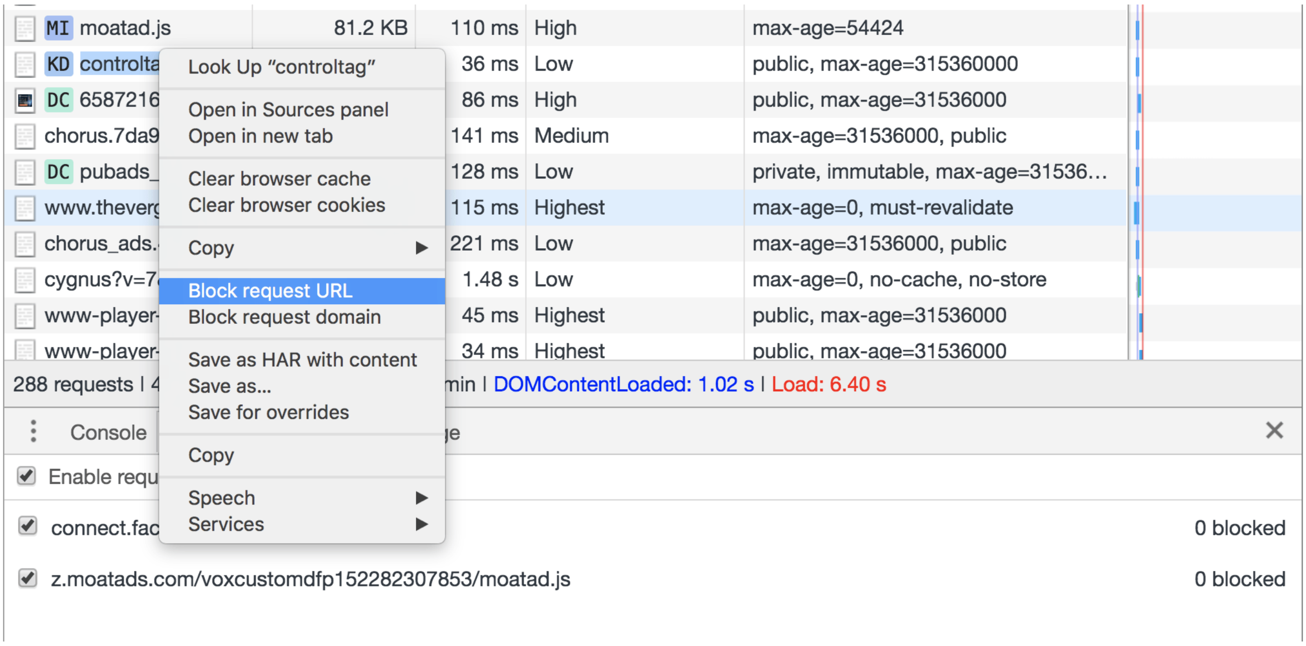
View Details

URL	Total Size	Transfer Time
…images/crop_skyline.png (treg.hearstnp.com)	436 KB	3,110 ms
…core/bridge3.193.1_en.html (imasdk.googleapis.com)	161 KB	1,150 ms
/wxwidget.loader.js?cid=370… (webwidgets.wsi.com)	146 KB	1,050 ms
/thumbs/AWcYwcDb-720.jpg (assets-jpcust.jwpsrv.com)	106 KB	760 ms
/libtrc/impl.297-104-RELEASE.js (cdn.taboola.com)	105 KB	750 ms
/thumbs/3W4TpC7t-720.jpg (assets-jpcust.jwpsrv.com)	101 KB	720 ms
…utils/nav_notch.svg (m.sfgate.com)	101 KB	720 ms
/thumbs/WI5hJrOT-720.jpg (assets-jpcust.jwpsrv.com)	93 KB	670 ms
/thumbs/g4o59IA0-720.jpg (assets-jpcust.jwpsrv.com)	91 KB	650 ms
/thumbs/NI0936Cl-720.jpg (assets-jpcust.jwpsrv.com)	87 KB	620 ms

Chrome DevTools Network Request Blocking

Chrome DevTools allows you to see how your page behaves when a particular script, stylesheet or other resource isn't available. This is done with network request blocking, a feature that can help measure the impact of blocking (dropping) specific third-party resources from your page.

To enable request blocking, right click on any request in the Network panel and select "Block Request URL". A Request blocking tab will display in the DevTools drawer, letting you manage which requests have been blocked.



MI moatad.js	81.2 KB	110 ms	High	max-age=54424
KD controlta		36 ms	Low	public, max-age=315360000
DC 6587216		86 ms	High	public, max-age=31536000
chorus.7da9		141 ms	Medium	max-age=31536000, public
DC pubads_		128 ms	Low	private, immutable, max-age=31536…
www.theverg		115 ms	Highest	max-age=0, must-revalidate
chorus_ads.		221 ms	Low	max-age=31536000, public
cygnus?v=7		1.48 s	Low	max-age=0, no-cache, no-store
www-player-		45 ms	Highest	public, max-age=31536000
www-player-		34 ms	Highest	public, max-age=31536000

288 requests | 4 … min | DOMContentLoaded: 1.02 s | Load: 6.40 s

☑ Enable requ…

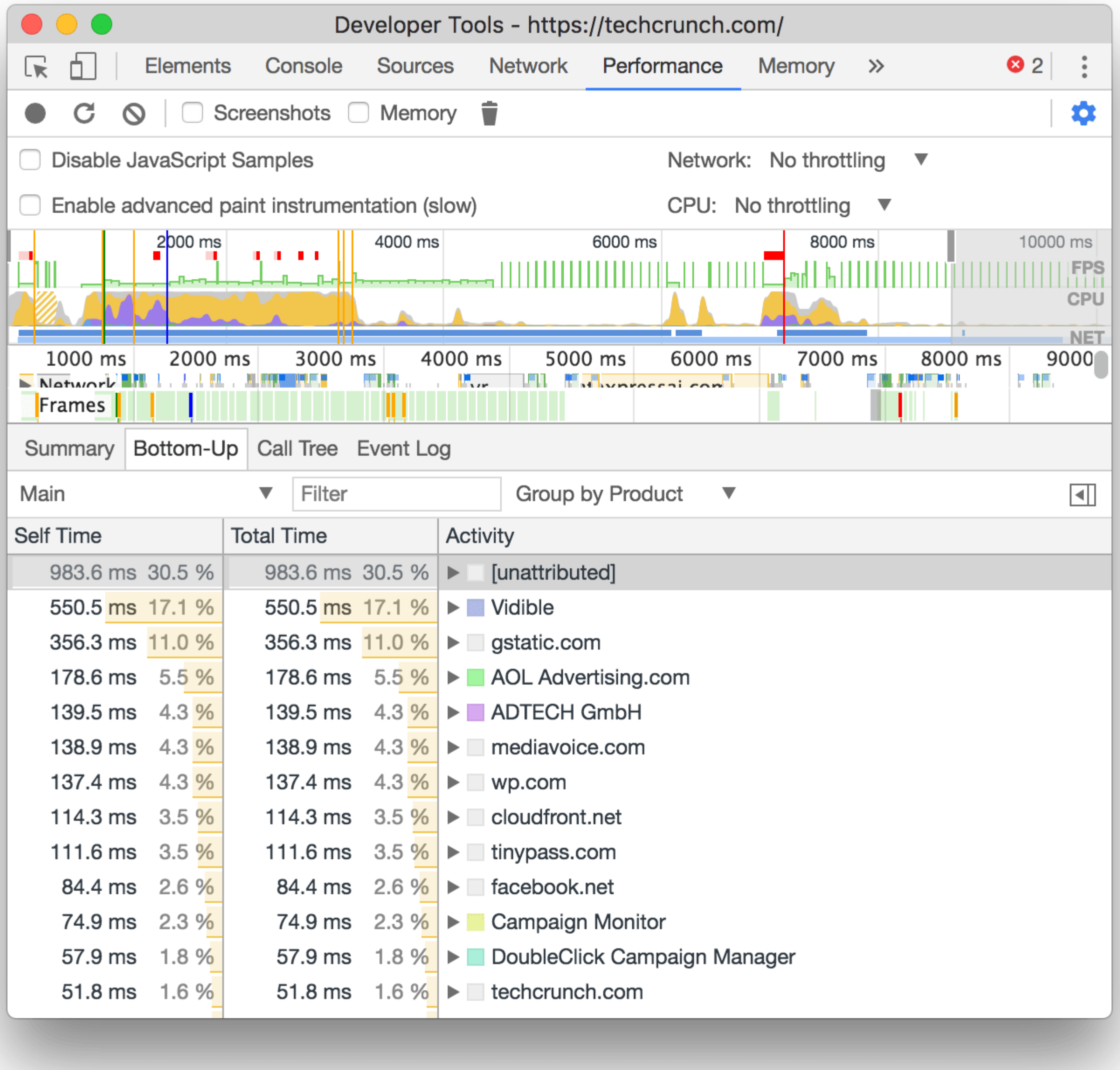
☑ connect.fac… 0 blocked

☑ z.moatads.com/voxcustomdfp152282307853/moatad.js 0 blocked

Chrome DevTools Performance Panel

The Performance panel in Chrome DevTools helps identify issues with your page's web performance. Clicking the record button and loading your page presents you with a waterfall representing where your site is spending time. At the bottom of the Performance panel, you will see a drawer starting with "Summary". Navigate to the “Bottom-up” tab.

Here, you can use the "Group by product" option in the Bottom-Up tab to group third-parties by the time they spent. This helps identify which third party products were the most costly. The Network panel also supports an option to highlight requests by product.



☐ Screenshots ☐ Memory

☐ Disable JavaScript Samples   Network: No throttling

☐ Enable advanced paint instrumentation (slow)   CPU: No throttling

Main   Group by Product

Self Time		Total Time		Activity
983.6 ms	30.5 %	983.6 ms	30.5 %	[unattributed]
550.5 ms	17.1 %	550.5 ms	17.1 %	Vidible
356.3 ms	11.0 %	356.3 ms	11.0 %	gstatic.com
178.6 ms	5.5 %	178.6 ms	5.5 %	AOL Advertising.com
139.5 ms	4.3 %	139.5 ms	4.3 %	ADTECH GmbH
138.9 ms	4.3 %	138.9 ms	4.3 %	mediavoice.com
137.4 ms	4.3 %	137.4 ms	4.3 %	wp.com
114.3 ms	3.5 %	114.3 ms	3.5 %	cloudfront.net
111.6 ms	3.5 %	111.6 ms	3.5 %	tinypass.com
84.4 ms	2.6 %	84.4 ms	2.6 %	facebook.net
74.9 ms	2.3 %	74.9 ms	2.3 %	Campaign Monitor
57.9 ms	1.8 %	57.9 ms	1.8 %	DoubleClick Campaign Manager
51.8 ms	1.6 %	51.8 ms	1.6 %	techcrunch.com

To learn more about how to analyze page load performance with the Chrome DevTools, see Get started with analyzing runtime performance.

A good **workflow** for measuring the impact of third-party scripts is:

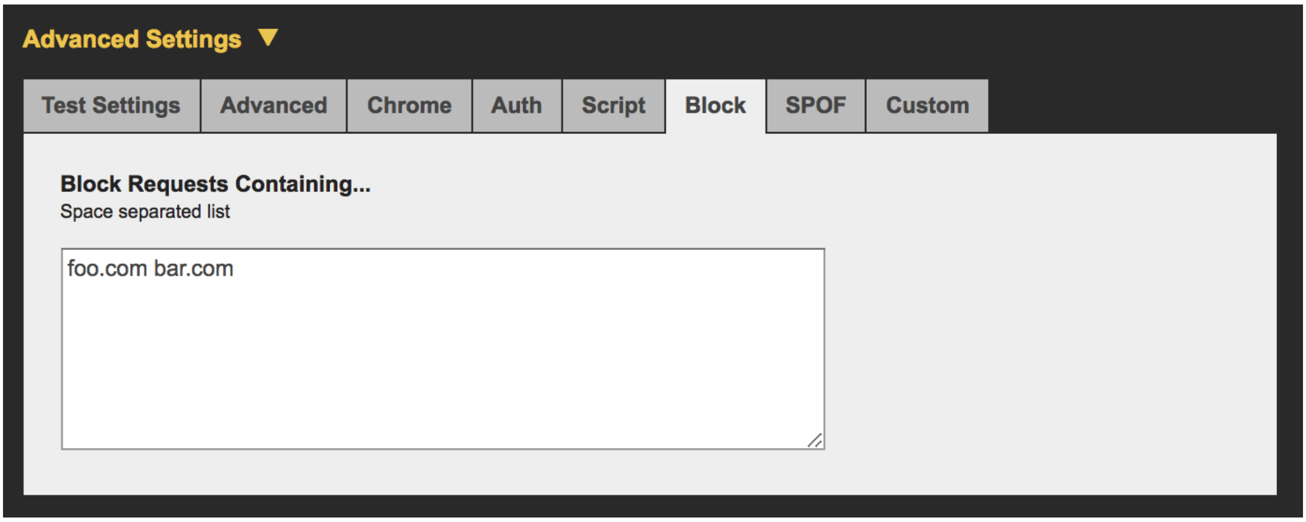
- Measure how long it takes to load your page using the Network panel.
    - To emulate real-world conditions, we recommend turning on network throttling and CPU throttling. On faster connections and desktop hardware, the impact of expensive scripts may not be as representative as it would on a mobile phone.
- Block the URLs or domains responsible for third-party scripts you believe are an issue (see *Chrome DevTools Performance Panel* for identifying costly scripts).
- Reload the page and re-measure how long the page takes without loading these blocked third-party scripts. You should hopefully see an improvement.

- There may be value in doing 3 or more runs of measurement and looking at the median for more stable figures. As third-party content can occasionally pull in different resources per page load, this could give you a more realistic spread. DevTools now supports multiple recordings in the performance panel, making this a little easier.

Measuring impact of Third-Party tags with WebPageTest

WebPageTest supports blocking individual requests from loading (which can be useful for blocking content like ads and third-party embeds) to measure their impact.

Under "Advanced Settings" is a Block tab. This can be used to specify a list of domains to block, simulating what it would be like if they didn't load at all.

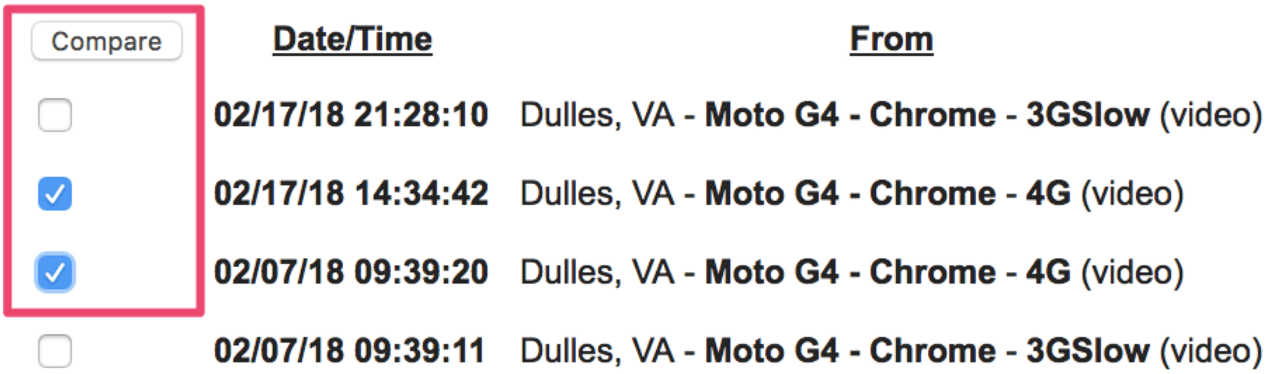


Block Requests Containing...
Space separated list

foo.com bar.com

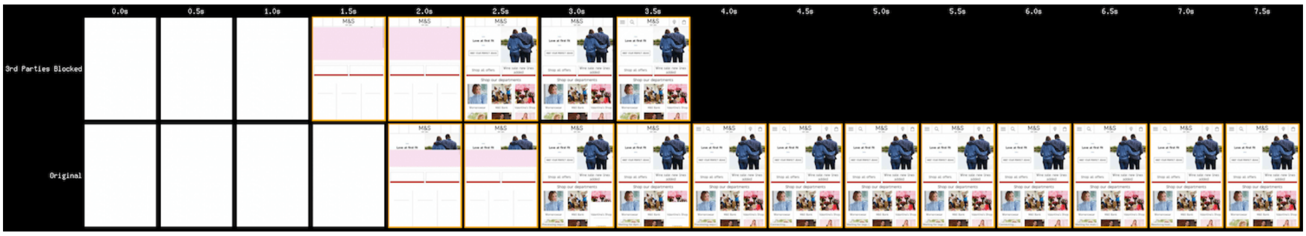
A workflow for using this feature is to:

- Test a page as normal
- Repeat the test with certain third-parties blocked
- Compare the two results (paying attention to the filmstrip). Results can be compared by selecting them from your Test History and clicking ‘Compare’.



Compare	Date/Time	From
☐	**02/17/18 21:28:10**	Dulles, VA - **Moto G4 - Chrome - 3GSlow** (video)
☑	**02/17/18 14:34:42**	Dulles, VA - **Moto G4 - Chrome - 4G** (video)
☑	**02/07/18 09:39:20**	Dulles, VA - **Moto G4 - Chrome - 4G** (video)
☐	**02/07/18 09:39:11**	Dulles, VA - **Moto G4 - Chrome - 3GSlow** (video)

Below we can see the difference between filmstrips both with and without third-party resources blocked. It can be useful to try this out for individual third-party origins to determine which ones have the biggest impact on your page-load performance:

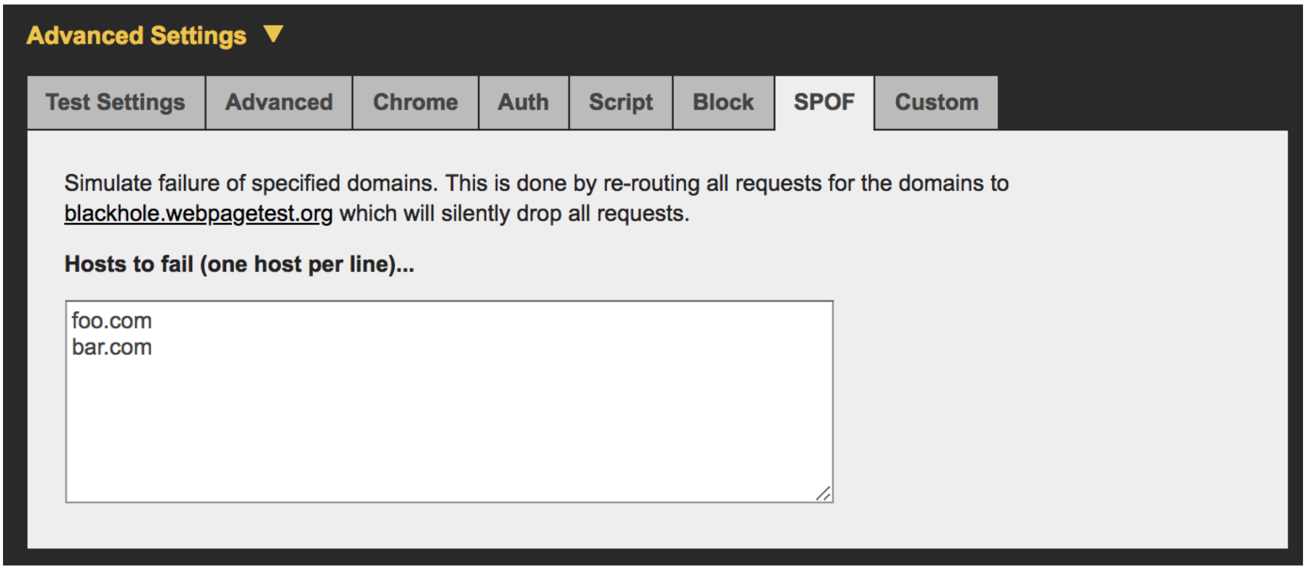


The impact of blocking third-party resources on a page using WPT's "block requests" feature from "Using WebPageTest To Measure The Impact Of Third-Party Tags" by Andy Davies

Note: WebPageTest also supports two commands operating at the DNS level for blocking domains. blockDomains - which takes a list of domains to block and blockDomainsExcept - takes a list of domains and blocks anything not on the list.

WebPageTest also has a single-point of failure (SPOF) tab. This allows you to simulate a timeout or complete failure to load a resource.

The difference between "SPOF" and "Block" is that SPOF slowly times out. This can make it useful for testing network resilience of third-party content to determine how well your pages hold up when services are under heavy load or temporarily unavailable.



Simulate failure of specified domains. This is done by re-routing all requests for the domains to blackhole.webpagetest.org which will silently drop all requests.

Hosts to fail (one host per line)...

```
foo.com
bar.com
```

Detecting expensive iframes using Long Tasks

When scripts in third-party iframes take a long time to run, they can block the main thread delaying other tasks from running. These long tasks can cause a negative user experience leading to sluggish event handlers or dropped frames.

To detect long tasks for Real User Monitoring (RUM), we can use the JavaScript PerformanceObserver API and observe longtask entries. As these entries contain an attribution property, we can track down which frame context was responsible for the task.

Below is an example that will log `longtask` entries to the console, including one for an "expensive" iframe:

```
<script>
    const observer = new PerformanceObserver((list) => {

        for (const entry of list.getEntries()) {

            // Attribution entry including "containerSrc":"https://example.com"
            console.log(JSON.stringify(entry.attribution));

        }

    });

    observer.observe({ entryTypes: ['longtask'] });
</script>

<!-- Imagine this is an iframe with expensive long tasks -->
<iframe src="https://example.com"></iframe>
```

To learn more about monitoring Long Tasks, read Phil Walton’s User-centric Performance Metrics.

How do you load third-party script efficiently?

If a third-party script is slowing down your page load, you have several options to improve performance:

- Load the script using the async or defer attribute to avoid blocking document parsing.
- Consider self-hosting the script if the third-party server is slow.
- Consider removing the script if it doesn't add clear value to your site.
- Consider Resource Hints like `<link rel=preconnect>` or `<link rel=dns-prefetch>` to perform a DNS lookup for domains hosting third-party scripts.

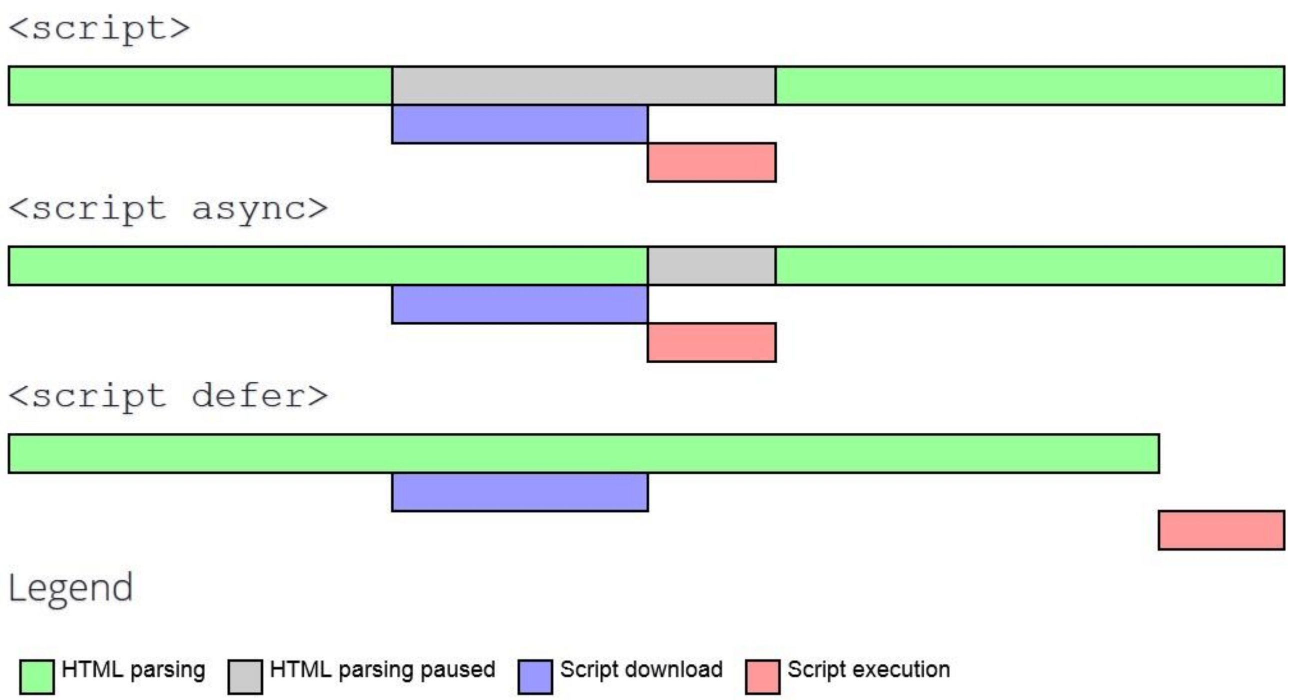
Use async or defer

JavaScript execution is parser blocking. This means when the browser encounters a script it must pause DOM construction, hand this over to the JavaScript engine and allow script execution before proceeding with DOM construction.

The async and defer attributes change this behavior.

- With async, the browser downloads the script asynchronously while it continues to parse the HTML document. When the script finishes downloading, parsing is blocked while the script executes.
- With defer, the browser downloads the script asynchronously while it continues to parse the HTML document. The script doesn't run until the parsing is complete.

If that's too many words, here's a pretty picture:



Credit: Growing with the web

In general, you should always use **async** or `defer` for third party scripts (unless the script does something necessary for the critical rendering path):

- Use **async** if it's important to have the script run earlier in the loading process. This might include some analytics scripts, for example.
- Use `defer` for less critical resources. A video player that's below-the-fold, for example.

Note: If performance is your primary concern, you could wait until your page has reached a key user moment (such as after the critical content has loaded) before adding async scripts. You should also take care not to `async` load libraries like jQuery just because they are coming from a third-party CDN.

Note: In Blink-based browsers, `async` and `defer` currently [lower the priority of the network request]() for resources so it can cause it to load significantly later than it would as a blocking script. This is useful to know in particular for analytics scripts.

Should you ever load third-party scripts without **async** or `defer`? You could make a case for this if the script is a crucial part of your site functionality. For example, if you're loading your main UI library or framework from a CDN, it will be badged as "third-party script" in DevTools, but should be treated as an essential part of your site, not an add-on.

Note that not all scripts work if loaded asynchronously. Check the docs for any third-party scripts you're using. If you're using a script that can't be loaded asynchronously, you might want to consider an alternative, or eliminating the script if possible. Some third parties may *highly recommend* to load their scripts sync (to get ahead of other scripts), even if they would work fine async so do due diligence when evaluating strategies for loading third-party scripts.

Note: `async` is not a silver bullet. If a marketing team wants to load a large number of tracking scripts on a page, this number will still introduce bottlenecks that can impact how soon users can engage with a page on load.

Use Resource Hints to reduce connection setup time

Establishing connections to third-party origins can take a significant amount of time - particularly on slow networks. Many steps can add up to delays including DNS lookups, redirects, and potentially several round trips to each third-party server to handle the request.

You can use Resource Hints like to perform a DNS lookup for domains hosting third-party scripts. When the request for them is finally made, time can be saved as the DNS lookup has already been carried out.

```
<link rel="dns-prefetch" href="http://example.com">
```

If the third-party domain you are referencing uses HTTPS, you may also consider as this will both perform the DNS lookup *and* resolve TCP round-trips and handle TLS negotiations. These other steps can be very slow as they involve looking at SSL certificates for verification, so consider Resource Hints seriously if you find third-party setup time to be an issue.

```
<link rel="preconnect" href="https://cdn.example.com">
```

"Sandbox" scripts with an iframe

There are cases where third-party scripts can be loaded directly into an iframe. By restricting such scripts to iframes, they won't block execution of the main page. This is the same approach that AMP takes to keeping JavaScript out of the critical path. Note that this approach will still block the `onload` event so try not to attach critical functionality to `onload`.

Note: Chrome is also exploring support for Feature Policy - a set of policies allowing a developer to selectively disable access to certain browser features. This can prevent third-party content introducing unwanted behaviors to a site.

Self-host third-party scripts

Self-hosting third-party scripts may be an option if you would like more control over a script's loading story. For example, if you wanted to reduce DNS or round-trip times, improve HTTP caching headers or take advantage of advanced techniques like HTTP/2 server push. Self-hosting may be a viable consideration if a script is considered critical.

Self-hosting can come with a number of big caveats:

- Scripts can go out of date. This can be a large issue as it prevents you from getting important security fixes without manually updating.
- Scripts that are self-hosted won't get automatic updates due to an API change. One example: a publisher with 90% of their revenue from ads discovers that ads didn't serve for half a day due to an API change that their self-hosted script didn't account for, leading to loss in income.

An alternative to self-hosting scripts would be using Service Workers to cache them. This can give you greater control over how often they are re-fetched from the network. This could also be used to create a loading strategy where requests for non-essential third parties are throttled until the page reaches a key user moment.

A/B Test smaller samples of users

A/B testing (or split-testing) is a technique for experimenting with two versions of a page to determine which one performs best. This is done by enabling both variants (A and B) for different samples of your website traffic. The page that provides a better conversion rate wins.

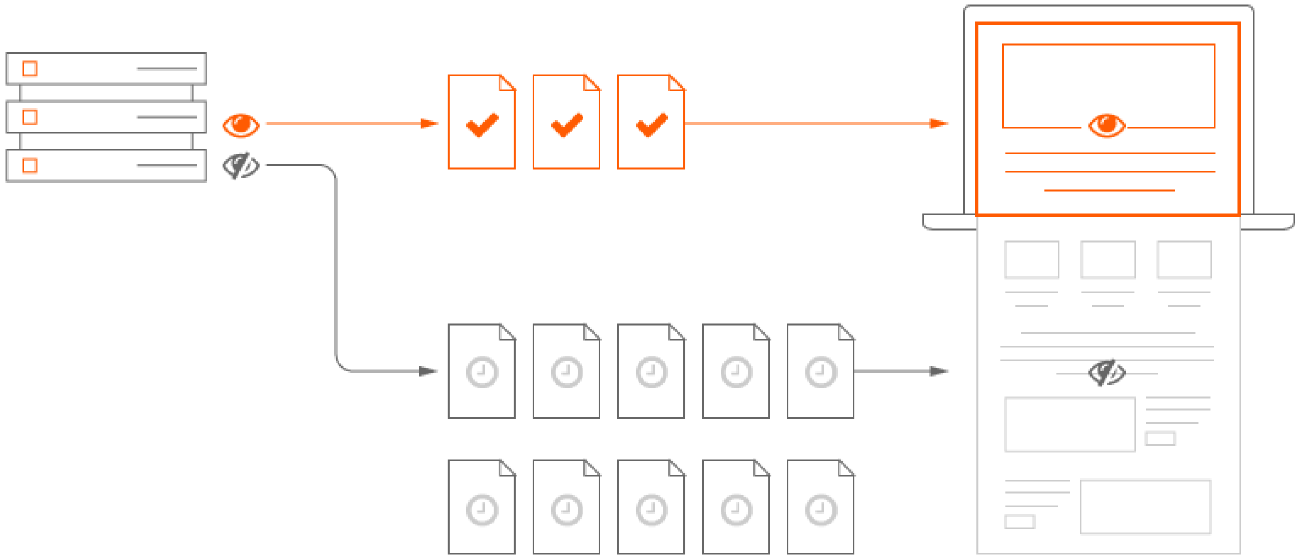
A/B testing is a very useful tool for analyzing user experience and behavior.

However, by design, A/B testing delays rendering to figure out which experiment needs to be active. JavaScript is often used to check if any of your users belong to an A/B test experiment and then enable the correct variant. This pattern can lead to 100% of your users being sent down large, costly script even if they don't belong to the sample receiving the experiment.

A good alternative in this case is to send A/B testing scripts for only a subset of your user base (e.g 10% vs 100%), ideally attempting to decide whether they belong in a test sample on the server-side. This improves the loading experience for the majority of users while still making split-testing possible.

Lazy-load Third Party Resources

Embedded third-party resources (such as ads or videos) can be a big contributor to slow page speed when constructed poorly. Lazy-loading can be used to only load embedded resources when necessary. For example, serving an ad in the footer only when a user scrolls down the page. Another pattern is lazy-loading content after the main page content loads but before a user might otherwise interact with the page.



Note: LazySizes is a popular JavaScript library for lazy-loading images and iframes. It supports YouTube embeds and widgets. Care does need to be taken when lazy-loading any resources as this technique is often powered by JavaScript and can be subject to issues on flaky network connections.

DoubleClick have guidance on how to lazy-load ads in their official documentation. If used properly, lazy loading can increase the overall viewability percentage of an ad. For example, MediaVine switched to lazy-loading ads and saw a 200% improvement in page load speed.

Efficient lazy-loading with Intersection Observer

Historically, solutions for detecting if an element is visible in the viewport (in order to lazy-load its content) have been error-prone, often causing the browser to become sluggish. Solutions have often listened for scroll or resize events, then used DOM APIs like getBoundingClientRect() to calculate where elements are relative to the viewport. This works, but is not efficient.

IntersectionObserver is a browser API that allows us to efficiently detect when an observed element enters or exits the browser's viewport. Learn more about how to use it for lazy-loading resources. LazySizes also has optional support for IntersectionObserver.

Analytics can be complicated

Analytics scripts should never slow down your page load experience, but if you defer the load too long you can miss valuable analytics data. Fortunately, there are some well-known patterns for initializing analytics lazily while retaining early page-load data.

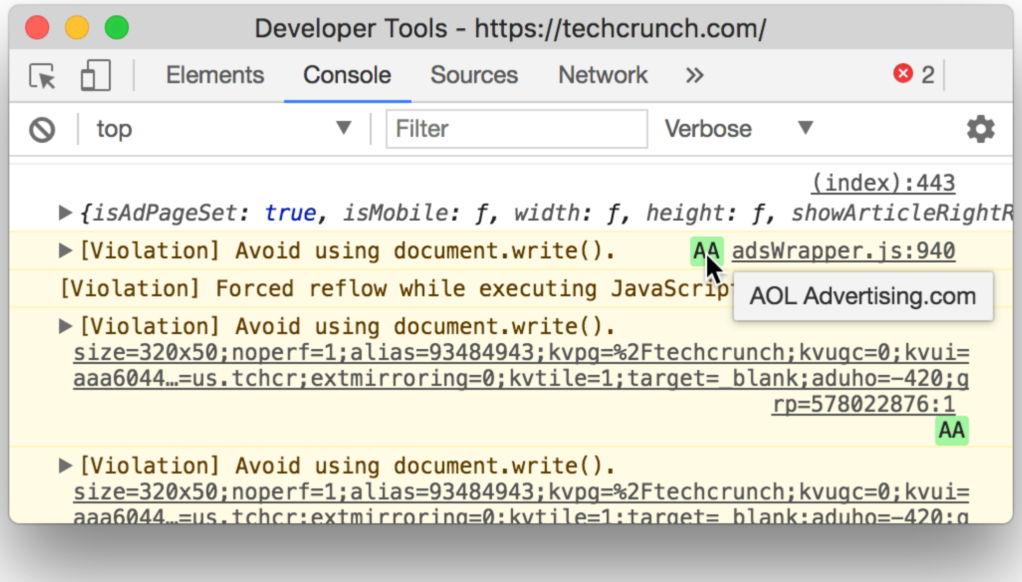
Phil Walton's blog post, The Google Analytics Setup I Use on Every Site I Build covers one such pattern for Google Analytics.

What patterns should I avoid with third-party script?

Avoid document.write()

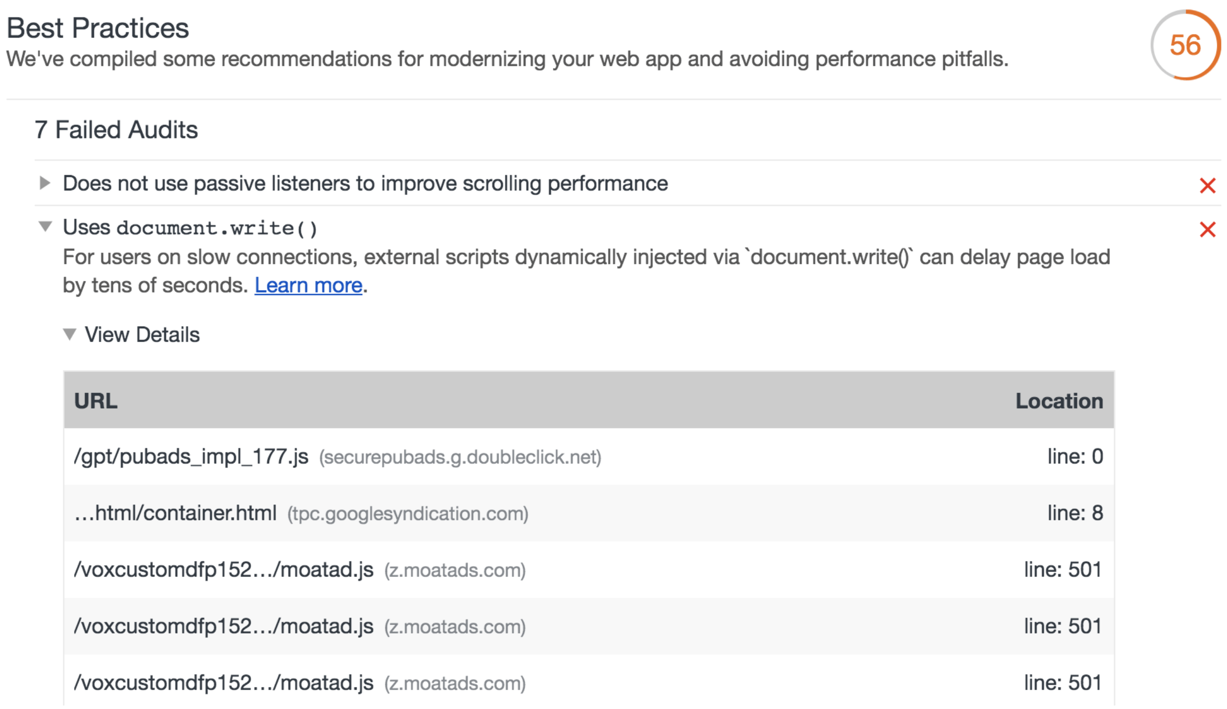
Third-party scripts sometimes use document.write() to inject and load scripts. This is particularly true of older services that haven’t been updated in some time. Thankfully, many third-parties offer an option to asynchronously load themselves, which allows third-party scripts to load without blocking the display of the rest of the content on the page.

The fix for document.write() is to simply not inject scripts using it. As of Chrome 53, Chrome DevTools will log warnings to the console for problematic use of document.write():



```
                                                        (index):443
▶ {isAdPageSet: true, isMobile: f, width: f, height: f, showArticleRightR
▶ [Violation] Avoid using document.write().         AA adsWrapper.js:940
[Violation] Forced reflow while executing JavaScrip  AOL Advertising.com
▶ [Violation] Avoid using document.write().
  size=320x50;noperf=1;alias=93484943;kvpg=%2Ftechcrunch;kvugc=0;kvui=
  aaa6044…=us.tchcr;extmirroring=0;kvtile=1;target=_blank;aduho=-420;g
                                                     rp=578022876:1
                                                                 AA
▶ [Violation] Avoid using document.write().
  size=320x50;noperf=1;alias=93484943;kvpg=%2Ftechcrunch;kvugc=0;kvui=
  aaa6044…=us.tchcr;extmirroring=0;kvtile=1;target=_blank;aduho=-420;g
```

To discover the use of document.write() at scale, you can check for HTTP headers sent to your browser when this intervention for Chrome happens. Lighthouse can also highlight any third-party scripts still using document.write() in the Lighthouse report:



Best Practices
We've compiled some recommendations for modernizing your web app and avoiding performance pitfalls.

7 Failed Audits

▶ Does not use passive listeners to improve scrolling performance ✕

▼ Uses `document.write()` ✕
For users on slow connections, external scripts dynamically injected via `document.write()` can delay page load by tens of seconds. Learn more.

▼ View Details

URL	Location
/gpt/pubads_impl_177.js (securepubads.g.doubleclick.net)	line: 0
…html/container.html (tpc.googlesyndication.com)	line: 8
/voxcustomdfp152…/moatad.js (z.moatads.com)	line: 501
/voxcustomdfp152…/moatad.js (z.moatads.com)	line: 501
/voxcustomdfp152…/moatad.js (z.moatads.com)	line: 501

Use Tag Managers but use them wisely

Note: Exercise caution when using GTM. Although it can minimize the overhead of third-party tags, it also makes it trivial for anyone with credentials to add costly tags.

A "tag" is a snippet of code that allows digital marketing teams to collect data, set cookies or integrate third-party content like social media widgets into a site. These tags have a cost to your page's loading performance - additional network requests, heavy JavaScript dependencies, images and resources the tag itself may pull in.

Managing these tags can become a real mess over time as marketing teams wish to add more ways to understand users and engineering tries to minimize the impact tags can have on user experience. To keep experiences fast, we recommend using a Tag manager. Tag managers:

- allow many pieces of third-party embed code to be managed from a single place (usually a user-interface)
- attempt to minimize how much third-party tags need to be deployed to sites.

Note: Even though individual tags can be asynchronously loaded, they still need to be read and executed individually. This could mean requesting more data while the page is still loading. Tag managers address this by reducing the number of calls a browser needs to make for them down to one.

Google Tag Manager (GTM) is one such popular tag manager:

"Google Tag Manager is an asynchronous tag, meaning that when it executes, it does not block other elements from rendering on the page. It also causes the other tags that are deployed via Google Tag Manager to be deployed asynchronously, meaning that a slow loading tag won't block other tracking tags."

Tag managers may improve page load performance by reducing how many calls to external resources are needed - as long as you **are not** pulling in a large number of tags. They also allow tags a way to collect values in a single unique place. For GTM, this is the Data Layer. If multiple third parties wish to trigger conversion-tracking data, they can do this by pulling from the Data Layer.

Risks when using Tag managers

When using a tag manager, great care needs to be taken to avoid slowing down how quickly pages load. This is because:

- Anyone with credentials and access can easily add not just more tags, but *any* JavaScript they want. Although tag managers can load tags asynchronously, this can still lead to an excess of costly HTTP requests being made and executed. This can be minimized by only allowing one user to publish versions.
- Anyone can configure too many tag manager auto-event listeners. Every auto-event listener needs to be executed & the more code and network requests there are, the longer it can take for a page to fully load. With our performance guidance encouraging that you respond to events within 50ms, every tag manager event listener added eats away at that goal.

Avoid scripts that pollute the global scope

Third-party scripts injected into the unknown can sometimes load a number of their own JavaScript dependencies. This can pollute the global scope and cause accidental breakage in pages.

There is also no guarantee that code loaded from a third-party will remain the same as what you saw during testing. New features can be pushed out by third parties at any time, potentially breaking your page. Self-testing, sub-resource integrity and securely transmitting third-party code (to reduce the risk of in-transit modifications) can help here.

Be sure to conduct careful audits of the third-party scripts you load to ensure they're being good actors.

Mitigation strategies

Adding third-party scripts to a page implies a level of trust in the origin. There are some strategies you can take to minimize their impact on performance and security:

- **HTTPS** is a must. Sites working over HTTPS shouldn't have third-parties working over HTTP. An HTTPS page that includes content fetched using HTTP is called a mixed content page and will run into Mixed Content warnings.
- Consider the **sandbox attribute** on iframes. From a security perspective, this allows you to restrict the actions available from the iframe. Restrictions include `allow-scripts`controlling whether the context can run scripts.
- Consider **Content Security Policy** (CSP). Through a HTTP header in your server's response, you can define behaviors that are trusted in the page. CSP can be used to detect and mitigate against the effects of certain attacks, such as Cross Site Scripting (XSS).

CSP is particularly powerful as it includes directives such as script-src that specifies what are valid, allowed sources for JavaScript. Below is an example of how this can be used in practice:

```
// Given this CSP header

Content-Security-Policy: script-src https://example.com/

// The following third-party script will not be loaded or executed

<script src="https://not-example.com/js/library.js"></script>
```

Conclusion

With sites relying on more third-party scripts than ever, it's paramount not to ignore third-party script performance. Good things you can do:

- Become familiar with some of the most effective third-party script optimization methods like only loading tags that support the async loading pattern.
- Understand how to identify and fix issues with third-party script loading. This can help you take back control of your page load performance.

Third-party script optimization should be followed by on-going real-time performance monitoring of your scripts and communication with your third-party providers. The web is evolving at a rapid pace and a script's locally observed performance gives no guarantees that it will perform as well in the future or in the wild.

Further reading

Performance and Resilience: Stress-Testing Third Parties

Adding interactivity with JavaScript

Potential dangers with Third-party Scripts

How 3rd Party Scripts can be performant citizens on the web

Why Fast Matters - CSS Wizardry

The JavaScript Supply Chain Paradox: SRI, CSP and Trust in Third Party Libraries

Third-party CSS isn't safe

With thanks to Kenji Baheux, Jeremy Wagner, Pat Meenan, Philip Walton, Jeff Posnick and Cheney Tsai for their reviews.

Previous
← JavaScript Start-up Optimization

Next
Web Font Optimization →

Last updated July 2, 2018.