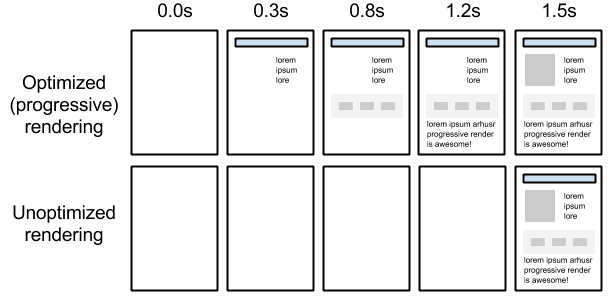
*Optimizing the critical rendering path* refers to prioritizing the display of content that relates to the current user action.

Delivering a fast web experience requires a lot of work by the browser. Most of this work is hidden from us as web developers: we write the markup, and a nice looking page comes out on the screen. But how exactly does the browser go from consuming our HTML, CSS, and JavaScript to rendered pixels on the screen?

Optimizing for performance is all about understanding what happens in these intermediate steps between receiving the HTML, CSS, and JavaScript bytes and the required processing to turn them into rendered pixels - that's the **critical rendering path**.



By optimizing the critical rendering path we can significantly improve the time to first render of our pages. Further, understanding the critical rendering path also serves as a foundation for building well-performing interactive applications. The interactive updates process is the same, just done in a continuous loop and ideally at 60 frames per second! But first, an overview of how the browser displays a simple page.

**Critical Rendering Path**

You will learn how to optimize any website for speed by diving into the details of how mobile and desktop browsers render pages.

You’ll learn about the Critical Rendering Path, or the set of steps browsers must take to convert HTML, CSS and JavaScript into living, breathing websites. From there, you’ll start exploring and experimenting with tools to measure performance and simple strategies to deliver the first pixels to the screen as early as possible. You’ll learn how to dive into recommendations from PageSpeed Insights and the Timeline view of Google Chrome’s Developer Tools to find the data you need to achieve immediate performance boosts!

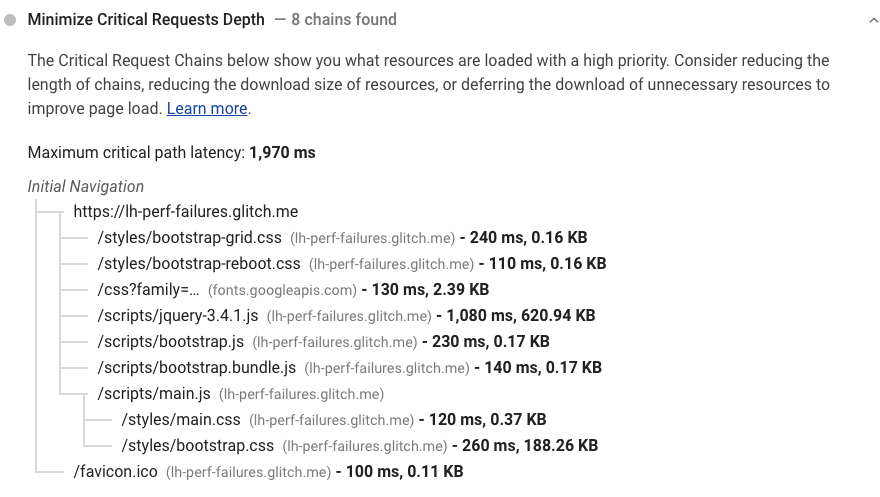
# Avoid chaining critical requests

May 2, 2019 • Updated Apr 29, 2020

Appears in: [Performance audits](https://web.dev/lighthouse-performance)

[Critical request chains](https://developers.google.com/web/fundamentals/performance/critical-rendering-path) are series of dependent network requests important for page rendering. The greater the length of the chains and the larger the download sizes, the more significant the impact on page load performance.

[Lighthouse](https://developers.google.com/web/tools/lighthouse/) reports critical requests loaded with a high priority:



See the [Lighthouse performance scoring](https://web.dev/performance-scoring) post to learn how your page's overall performance score is calculated.

## How Lighthouse identifies critical request chains [#](https://web.dev/critical-request-chains/?utm_source=lighthouse&utm_medium=devtools#how-lighthouse-identifies-critical-request-chains)

Lighthouse uses network priority as a proxy for identifying render-blocking critical resources. See Google's [Chrome Resource Priorities and Scheduling](https://docs.google.com/document/d/1bCDuq9H1ih9iNjgzyAL0gpwNFiEP4TZS-YLRp_RuMlc/edit) for more information about how Chrome defines these priorities.

Data on critical request chains, resource sizes, and time spent downloading resources is extracted from the [Chrome Remote Debugging Protocol](https://github.com/ChromeDevTools/devtools-protocol).

## How to reduce the effect of critical request chains on performance [#](https://web.dev/critical-request-chains/?utm_source=lighthouse&utm_medium=devtools#how-to-reduce-the-effect-of-critical-request-chains-on-performance)

Use the critical request chains audit results to target the resources that have the biggest effect on page load first:

* Minimize the number of critical resources: eliminate them, defer their download, mark them as async, and so on.
* Optimize the number of critical bytes to reduce the download time (number of round trips).
* Optimize the order in which the remaining critical resources are loaded: download all critical assets as early as possible to shorten the critical path length.

# **Current State**

As of June 2016, the table below represents how all resources in Chrome are handled:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Layout-blocking | Load in layout-blocking phase | Load one-at-a-time in layout-blocking phase | | |
| **Blink Priority** | **VeryHigh** | **High** | **Medium** | **Low** | **VeryLow** |
| **DevTools Priority** | **Highest** | **High** | **Medium** | **Low** | **Lowest** |
|  | Main Resource |  |  |  |  |
|  | CSS\*\*\* (early\*\*) |  | CSS\*\*\* (late\*\*) |  | CSS (mismatch) |
|  |  | Script (early\*\* or not from preload scanner) | Script (late\*\*) | Script (async) |  |
|  | Font | Font (preload) |  |  |  |
|  |  | Import |  |  |  |
|  |  | Image (in viewport - after layout) |  | Image |  |
|  |  |  |  | Media |  |
|  |  |  |  | SVG Document |  |
|  |  |  |  |  | Prefetch |
|  |  | Preload\* |  |  |  |
|  |  | XSL |  |  |  |
|  | XHR (sync) | XHR/fetch\* (async) |  |  |  |
|  |  |  | Favicon |  |  |

\* Preload using “as” or fetch using “type” use the priority of the type they are requesting. (e.g. preload as=stylesheet will use Highest priority). With no “as” they will behave like an XHR.

\*\* “Early” is defined as being requested before any non-preloaded images have been requested (“late” is after).

\*\*\* CSS where the media type doesn’t match is not fetched by the preload scanner and is only processed when the main parser reaches it which usually means it will be fetched very late and with a “late” priority.

# **Priority Changes**

Images always start at a low priority. If, at layout time, an image is discovered to be within the viewport then the priority will be boosted to High though that can be pretty late in the loading process and changing priority dynamically may have no impact if the request was already sent.

Late-body blocking scripts start at a medium priority but if the main HTML parser reaches them and gets blocked then the priority will be boosted to High.

Dev tools displays the final priority that was used for a given resource by the time it finished loading. If an image starts at low priority and is boosted to high, it will appear as high priority even if it ended up being delayed.

# **Net Stack Priority Names**

The Chrome networking stack uses the same 5 priority levels as the rest of Chrome but the names are slightly different and offset from each other. The main cases where you will encounter the networking stack version of the names is if you are working on the Chrome code itself or looking at a netlog. The net priority names are usually fully uppercased which makes it a bit easier to distinguish.  The full mapping is:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Net Priority | **HIGHEST** | **MEDIUM** | **LOW** | **LOWEST** | **IDLE** |
| Priority | **Highest** | **High** | **Medium** | **Low** | **Lowest** |

# **Priority Hints (proposal/experiment)**

The default priority behaviours above work well in the general case but there are times when a developer would benefit from being able to change the request ordering. For example, to signal that an image is important before layout discovers it is in the viewport or to signal that some async scripts are more important or less important.

The [priority hints spec](https://wicg.github.io/priority-hints/) gives developers a way to signal “high”, “low” or “auto” (default). The table below is what will be experimented with in Chrome where that “auto” is the same as the current defaults and some resource types allow for tweaking priority through high/low which will mostly map directly to Blink’s high/low priority levels.

Resource fetches at the same priority level are prioritized in the order they are discovered.

◉ : importance=”auto”

⬆ : importance=”high”

⬇ : importance=”low”

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | **Layout- blocking** | **Load in layout- blocking phase** | **Load one-at-a-time in layout-blocking phase** | | |
| **Blink Priority** | VeryHigh | High | Medium | Low | VeryLow |
| **DevTools Priority** | Highest | High | Medium | Low | Lowest |
| Main Resource | ◉ |  |  |  |  |
| CSS\*\*\* (early\*\*) | ⬆◉ | ⬇ |  |  |  |
| CSS\*\*\* (late\*\*) |  | ⬆ | ◉ | ⬇ |  |
| Script (early\*\* or not from preload scanner) |  | ⬆◉ |  | ⬇ |  |
| Script (late\*\*) |  | ⬆ | ◉ | ⬇ |  |
| Script (async/defer) |  | ⬆ |  | ◉⬇ |  |
| Font | ◉ |  |  |  |  |
| Font (preload) |  | ⬆◉ |  | ⬇ |  |
| Import |  | ◉ |  |  |  |
| Image (in viewport - after layout) |  | ⬆◉ |  | ⬇ |  |
| Image |  | ⬆ |  | ◉⬇ |  |
| Media (video/audio) |  | ⬆ |  | ◉⬇ |  |
| SVG Document |  | ⬆ |  | ◉⬇ |  |
| XHR (sync) - deprecated | ◉ |  |  |  |  |
| XHR/fetch\* (async) |  | ⬆◉ |  | ⬇ |  |
| Preload\* |  | ⬆◉ |  | ⬇ |  |
| Prefetch |  |  |  |  | ◉ |
| Favicon |  |  | ◉ |  |  |
| XSL |  | ◉ |  |  |  |

\* Preload using “as” or fetch using “type” use the priority of the type they are requesting unless identified above (like font). (e.g. preload as=stylesheet will use Highest priority). With no “as” they will behave like an XHR.

\*\* “Early” is defined as being requested before any non-preloaded images have been requested (“late” is after).

\*\*\* CSS where the media type doesn’t match is not fetched by the preload scanner and is only processed when the main parser reaches it which usually means it will be fetched very late and with a “late” priority.