Understand the critical path

The critical rendering path refers to the steps involved until the web page starts rendering in the browser. To render pages, browsers need the HTML document itself as well as all the critical resources necessary for rendering that document.

Getting the HTML document to the browser was covered by the previous General HTML performance considerations module (/learn/performance/general-html-performance). In this module, however, we'll look more at what the browser does *after* it downloads the HTML document in order to render the page.

Progressive rendering

The web is distributed by nature. Unlike native applications that are installed before use, browsers cannot depend on websites having all the resources necessary to render the page. Therefore, browsers are very good at rendering pages progressively. Native apps typically have an install phase, and then a running phase. However, for web pages and web apps, the lines between these two phases are much less distinct, and browsers have been specifically designed with that in mind.

Once the browser has resources to render a page, it will typically begin to do so. The choice therefore becomes about *when* to render: when is too early?

If the browser renders as soon as possible when it just has some HTML—but before it has any CSS or necessary JavaScript—then the page will momentarily look broken and change considerably for the final render. This is a worse experience than initially presenting a blank screen for a time until the browser has more of these resources needed for an initial render that offers a better user experience.

On the other hand, if the browser waits for *all* resources to be available instead of doing any sequential rendering, then the user will be left waiting for a long time; often unnecessarily so if the page was usable at a much earlier point in time.

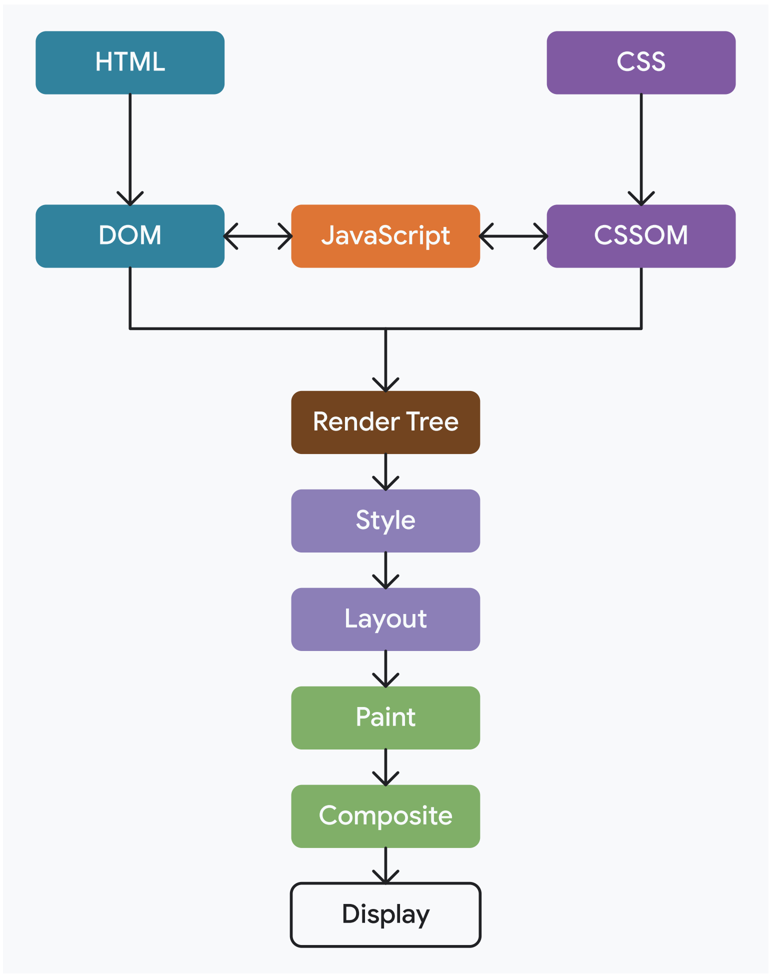
The browser needs to know what the minimum number of resources it should wait for in order to avoid presenting an obviously broken experience. On the other hand, the browser also shouldn't wait longer than necessary before presenting the user with some content. The sequence of steps the browser takes before performing that initial render is known as the *critical rendering path*.

Understanding the critical rendering path can help improve web performance by ensuring you don't block initial page rendering any more than necessary. At the same time, however, it's important to also not allow rendering to happen too early by removing the necessary resources for that initial render from the critical rendering path.

The (critical) rendering path

The rendering path involves the following steps:

- Constructing the Document Object Model (DOM) from the HTML.
- Constructing the CSS Object Model (CSSOM) from the CSS.
- Applying any JavaScript that alters the DOM or CSSOM.
- Constructing the render tree from the DOM and CSSOM.
- Perform style and layout operations on the page to see what elements fit where.
- Paint the pixels of the elements in memory.
- Composite the pixels if any of them overlap.
- Physically draw all the resulting pixels to screen.



The rendering process, as detailed in the previous list.

Only after all these steps have been completed, will the user see content on the screen.

Learn more: You can find more details about each of these steps in this critical rendering path article (/articles/critical-rendering-path) and its linked modules—and it can get even more complicated than those steps suggest (https://developer.chrome.com/articles/renderingng-architecture). However, for an initial understanding of web performance, this high-level overview should suffice.

This rendering process happens multiple times. The initial render invokes this process, but as more resources that affect the page's rendering become available, the browser will re-run this process—or maybe just parts of it—to update what the user sees. The critical rendering path focuses on the process previously outlined for the initial render, and depends on the critical resources necessary for it.

What resources are on the critical rendering path?

The browser needs to wait for some critical resources to download before it can complete the initial render. These resources include:

- Part of the HTML.
- Render-blocking CSS in the `<head>` element.
- Render-blocking JavaScript in the `<head>` element.

A key point is that the browser processes HTML in a streaming fashion. As soon as the browser gets any portion of a page's HTML, the browser starts processing it. The browser can then—and often does—decide to render it well before receiving the rest of a page's HTML.

Importantly, for the initial render, the browser will *not* typically wait for:

- All of the HTML.
- Fonts.
- Images.
- Non-render-blocking JavaScript outside of the `<head>` element (for example, `<script>` elements placed at the end of the HTML).
- Non-render-blocking CSS outside of the `<head>` element, or CSS with a `media` attribute (https://developer.mozilla.org/docs/Web/HTML/Element/link#conditionally_loading_resources_with_media_queries) value that does not apply to the current viewport.

Fonts and images are often regarded by the browser as content that to be filled in during subsequent page rerenders, so they don't need to hold up the initial render. This can, however, mean that areas of blank space are left in the initial render while text is hidden waiting on fonts, or until images are available. Worse still is when sufficient space is not reserved for certain types of content—particularly when image dimensions are not provided in the HTML—the page's layout can shift when this content loads later on. This aspect of the user experience is measured by the Cumulative Layout Shift (CLS) (/articles/cls) metric.

The `<head>` element is key to processing the critical rendering path. So much so that the next section covers it in some detail (/learn/performance/optimize-resource-loading). Optimizing the `<head>` element's contents is a key aspect of web performance. To understand the critical rendering path for now, though, you only need to know that the `<head>` element contains metadata about the page and its resources, but no actual content the user can see. Visible content is contained within the `<body>` element that follows the `<head>` element. Before the browser can render any content, it needs *both* the content to render as well as the metadata about how to render it.

However, not all resources referenced in the `<head>` element are strictly necessary for the initial page render, so the browser only waits for those that are. To identify which resources are in the critical rendering path, you need to understand render-blocking and parser-blocking CSS and JavaScript.

Render-blocking resources

Some resources are deemed so critical that the browser pauses page rendering until it has dealt with them. CSS falls into this category by default.

When a browser sees CSS—whether it's inline CSS in a `<style>` element, or an externally referenced resource specified by a `<link rel=stylesheet href="...">` element—the browser avoids rendering any more content until it has completed downloading and processing that CSS.

Note: Although CSS is render-blocking by default, it can be turned into a non-render-blocking resource by changing the `<link>` element's `media` attribute to specify a value that doesn't match the current conditions: `<link rel=stylesheet href="..." media=print>`. This has been used in the past (https://www.filamentgroup.com/lab/load-css-simpler/) to allow non-critical CSS to load in a non-render blocking fashion.

Just because a resource blocks rendering does not necessarily mean it stops the browser from doing anything else. Browsers try to be as efficient as possible, so when a browser sees it needs to download a CSS resource, it will request it and pause *rendering*, but will still carry on *processing* the rest of the HTML and look for other work to do in the meantime.

Render-blocking resources, like CSS, used to block all rendering of the page when they were discovered. This means that whether some CSS is render-blocking or not depends on whether the browser has discovered it. Some browsers (Firefox initially (https://jakearchibald.com/2016/link-in-body/), and now also Chrome (https://chromestatus.com/feature/5696805480169472)) only block rendering of content below the render-blocking resource. This means that for the critical render-blocking path, we're typically interested in render-blocking resources in the `<head>`, as they effectively block rendering of the entire page.

A more recent innovation is the `blocking=render` attribute (https://html.spec.whatwg.org/multipage/urls-and-fetching.html#blocking-attributes), added to Chrome

105 (https://chromestatus.com/feature/5452774595624960). This allows developers to explicitly mark a `<link>`, `<script>` or `<style>` element as rendering-blocking until the element is processed, but still allowing the parser to continue processing the document in the meantime.

Parser-blocking resources

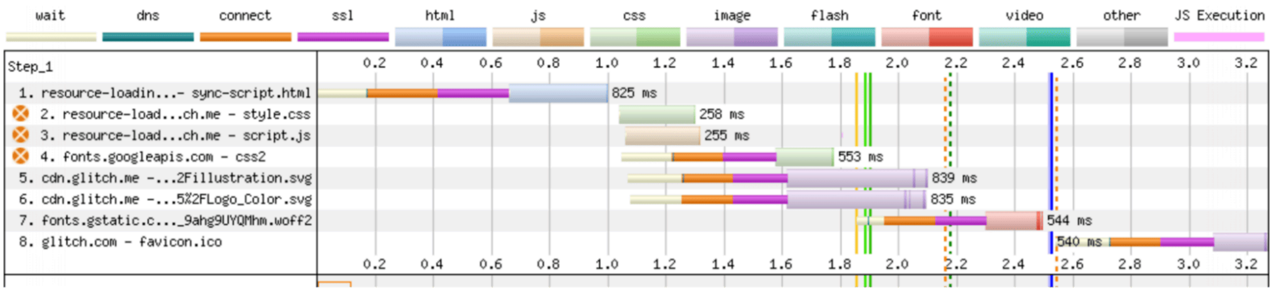
Parser-blocking resources are those that prevent the browser from looking for other work to do by continuing to parse the HTML. JavaScript by default is parser-blocking (unless specifically marked as asynchronous (https://developer.mozilla.org/docs/Web/HTML/Element/script#async) or deferred (https://developer.mozilla.org/docs/Web/HTML/Element/script#defer)), as JavaScript can change the DOM or the CSSOM upon its execution. Therefore, it's not possible for the browser to continue processing other resources until it knows the full impact of the requested JavaScript on a page's HTML. Synchronous JavaScript therefore blocks the parser.

Parser-blocking resources are effectively render-blocking as well. Since the parser can't continue past a parsing-blocking resource until it has been fully processed, it can't access and render the content after it. The browser can render any HTML received so far while it waits, but where the critical rendering path is concerned, any parser-blocking resources in the `<head>` effectively mean that all page content is blocked from being rendered.

Blocking the parser can have a huge performance cost—much more than just blocking rendering. For this reason, browsers will try to reduce this cost by using a secondary HTML parser known as the preload scanner (/articles/preload-scanner) to download upcoming resources while the primary HTML parser is blocked. While not as good as actually parsing the HTML, it does at least allow the networking functions in the browser to work ahead of the blocked parser, meaning it will be less likely to be blocked again in the future.

Identify blocking resources

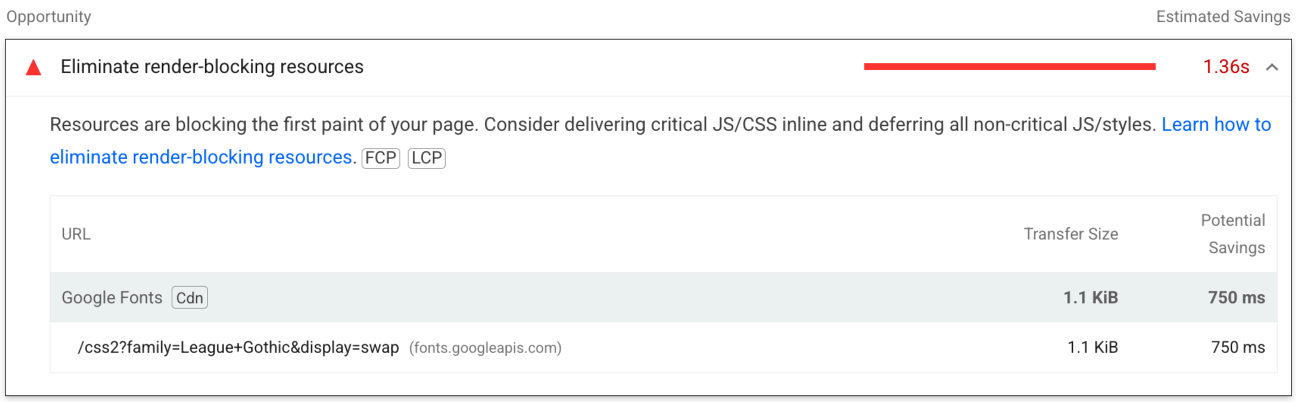
Many performance auditing tools identify render and parser-blocking resources. WebPageTest (https://www.webpagetest.org/) marks render-blocking resources with an orange circle to the left of the resource's URL:



Network waterfall diagram generated by WebPageTest.

All render-blocking resources need to be downloaded and processed before rendering can start, which is noted by the solid dark green line in the waterfall.

Lighthouse also highlights render-blocking resources, but in a more subtle way, and only if the resource actually delays page rendering. This can be helpful to avoid false positives where you are otherwise minimizing render-blocking. Running the same page URL as the preceding WebPageTest figure through Lighthouse only identifies one of the stylesheets as a rendering-blocking resource.


Opportunity Estimated Savings

Eliminate render-blocking resources 1.36s

Resources are blocking the first paint of your page. Consider delivering critical JS/CSS inline and deferring all non-critical JS/styles. Learn how to eliminate render-blocking resources. FCP LCP

URL	Transfer Size	Potential Savings
Google Fonts Cdn	**1.1 KiB**	**750 ms**
/css2?family=League+Gothic&display=swap (fonts.googleapis.com)	1.1 KiB	750 ms

Lighthouse's audit for eliminating render-blocking resources.

Optimize the critical rendering path

Optimizing the critical rendering path involves reducing the time to receive the HTML (represented by the Time to First Byte (TTFB) metric (/articles/ttfb)) as detailed in the previous module, and reducing the impact of render-blocking resources. These concepts are investigated in the next modules.

The critical contentful rendering path

For a long time, the critical rendering path has concerned itself with the initial render. However, more user-centric metrics (/articles/user-centric-performance-metrics) for web performance have