

9 November, 2018

CSS and Network Performance

Despite having been called *CSS Wizardry* for over a decade now, there hasn't been a great deal of CSS-related content on this site for a while. Let me address that by combining my two favourite topics: CSS and performance.

CSS is critical to rendering a page—a browser will not begin rendering until all CSS has been found, downloaded, and parsed—so it is imperative that we get it onto a user's device as fast as we possibly can. Any delays on the Critical Path affect our Start Render and leave users looking at a blank screen.

What's the Big Problem?

Broadly speaking, this is why CSS is so key to performance:

1. A browser can't render a page until it has built the Render Tree;
2. the Render Tree is the combined result of the DOM and the CSSOM;
3. the DOM is HTML plus any blocking JavaScript that needs to act upon it;
4. the CSSOM is all CSS rules applied against the DOM;
5. it's easy to make JavaScript non-blocking with `async` and `defer` attributes;
6. making CSS asynchronous is much more difficult;
7. so a good rule of thumb to remember is that **your page will only render as quickly as your slowest stylesheet**

With this in mind, we need to construct the DOM and CSSOM as quickly as possible. Constructing the DOM is, for the most part, relatively fast: your first HTML response *is* the DOM. However, as CSS is almost always a subresource of the HTML, constructing the CSSOM usually takes a good deal longer.

In this post I want to look at how CSS can prove to be a substantial bottleneck on the network (both in itself and for other resources) and how we can mitigate it, thus shortening the Critical Path and reducing our time to Start Render.

Employ Critical CSS

If you are able, one of the most effective ways to cut down the time to Start Render is to make use of the Critical CSS pattern: identify all of the styles needed for Start Render (commonly the styles needed for everything above the fold), inline them in `<style>` tags in the `<head>` of your document, and asynchronously load the remaining stylesheet off of the Critical Path.

While this strategy is effective, it's not simple: highly dynamic sites can be difficult to extract styles from, the process needs to be automated, we have to make assumptions about what *above the fold* even is, it's hard to capture edge cases, and tooling still in its relative infancy. If you're working with a large or legacy codebase, things get even more difficult...

Split Your Media Types

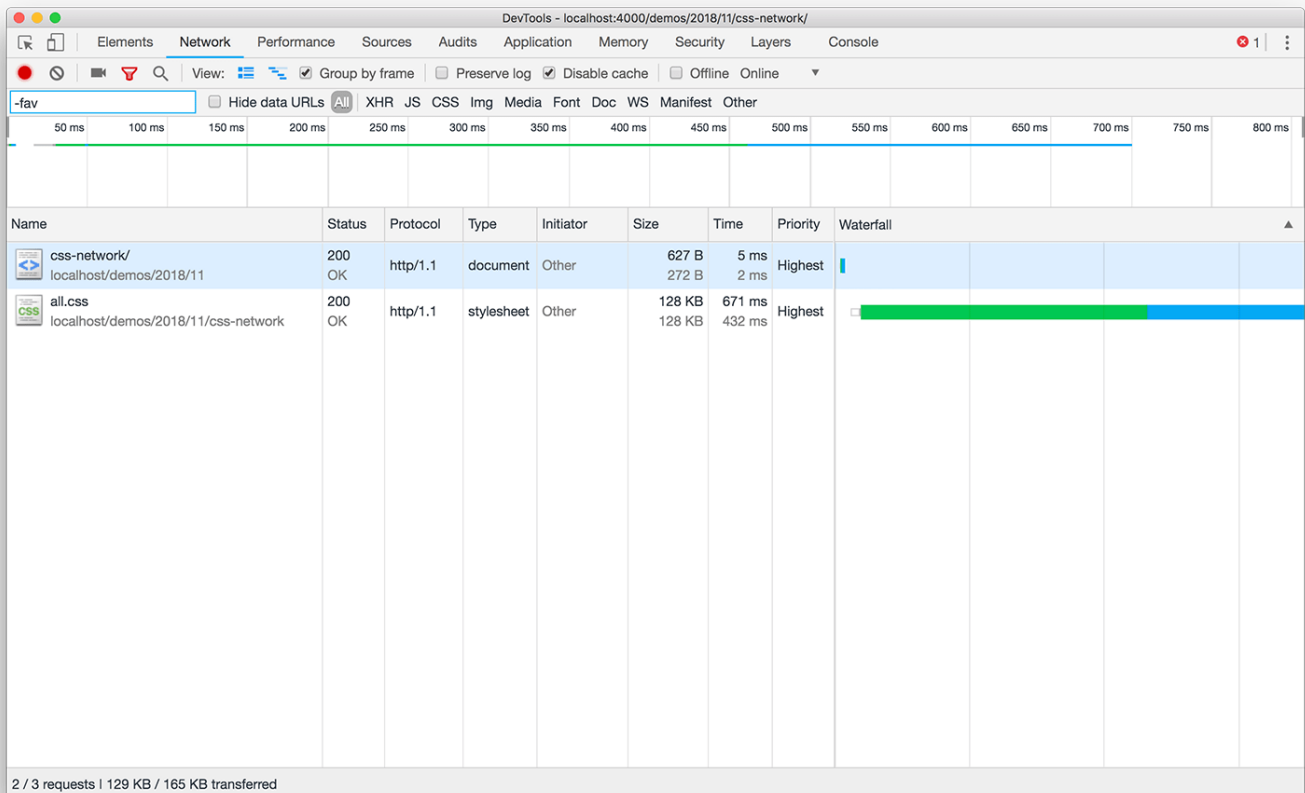
So if achieving Critical CSS is proving quite tricky—and it probably is—another option we have is to split our main CSS file out into its individual Media Queries. The practical upshot of this is that the browser will...

- download any CSS needed for the current context (medium, screen size, resolution, orientation, etc.) with a very high priority, blocking the Critical Path, and;
- download any CSS not needed for the current context with a very low priority, completely off of the Critical Path.

Basically, any CSS not needed to render the current view is effectively lazyloaded by the browser.

```
<link rel="stylesheet" href="all.css" />
```

If we're bundling all of our CSS into one file, this is how the network treats it:

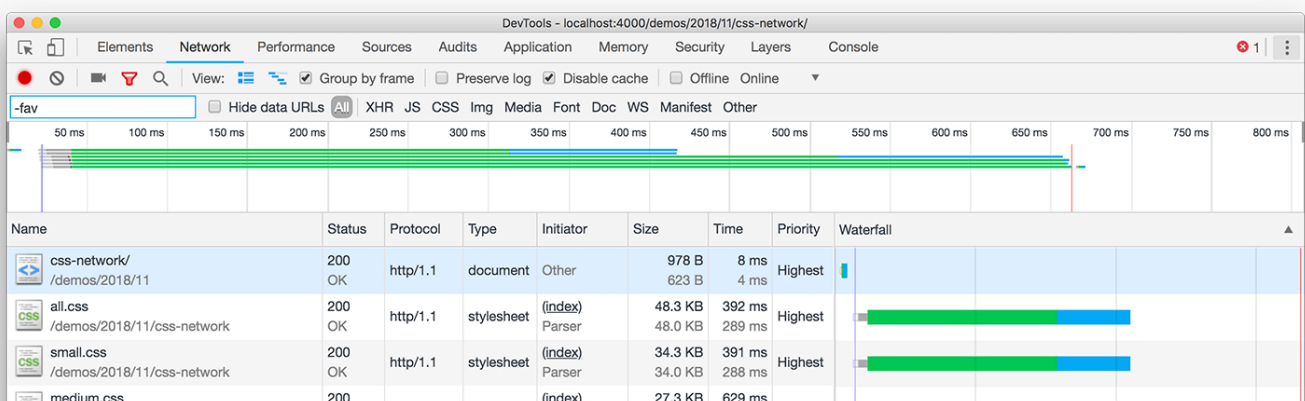







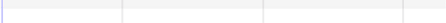


Notice the single CSS file carries a *Highest* priority.

If we can split that single, all-render blocking file into its respective Media Queries:

```
<link rel="stylesheet" href="all.css" media="all" />
<link rel="stylesheet" href="small.css" media="(min-width: 20em)" />
<link rel="stylesheet" href="medium.css" media="(min-width: 64em)" />
<link rel="stylesheet" href="large.css" media="(min-width: 90em)" />
<link rel="stylesheet" href="extra-large.css" media="(min-width: 120em)" />
<link rel="stylesheet" href="print.css" media="print" />
```

Then we see that the network treats files differently:



	/demos/2018/11/css-network	OK	http/1.1	stylesheet	Parser	27.0 KB	490 ms	Lowest	
	large.css	200	http/1.1	stylesheet	(index) Parser	11.3 KB	632 ms	Lowest	
	extra-large.css	200	http/1.1	stylesheet	(index) Parser	5.3 KB	631 ms	Lowest	
	print.css	200	http/1.1	stylesheet	(index) Parser	3.3 KB	633 ms	Lowest	

7 / 8 requests | 131 KB / 167 KB transferred | Finish: 670 ms | DOMContentLoaded: 21 ms | Load: 662 ms

CSS files not required to render the current context are assigned a *Lowest* priority.

The browser will still download all of the CSS files, but it will only block rendering on files needed to fulfil the current context.

Avoid @import in CSS Files

The next thing we can do to help Start Render is much, much simpler. **Avoid using @import in your CSS files.**

@import, by virtue of how it works, is slow. It's really, really bad for Start Render performance. This is because we're actively creating more roundtrips on the Critical Path:

1. Download HTML;
2. HTML requests CSS;
 - (Here's where we'd like to be able to construct the Render Tree, but;)
3. CSS requests more CSS;
4. build the Render Tree.

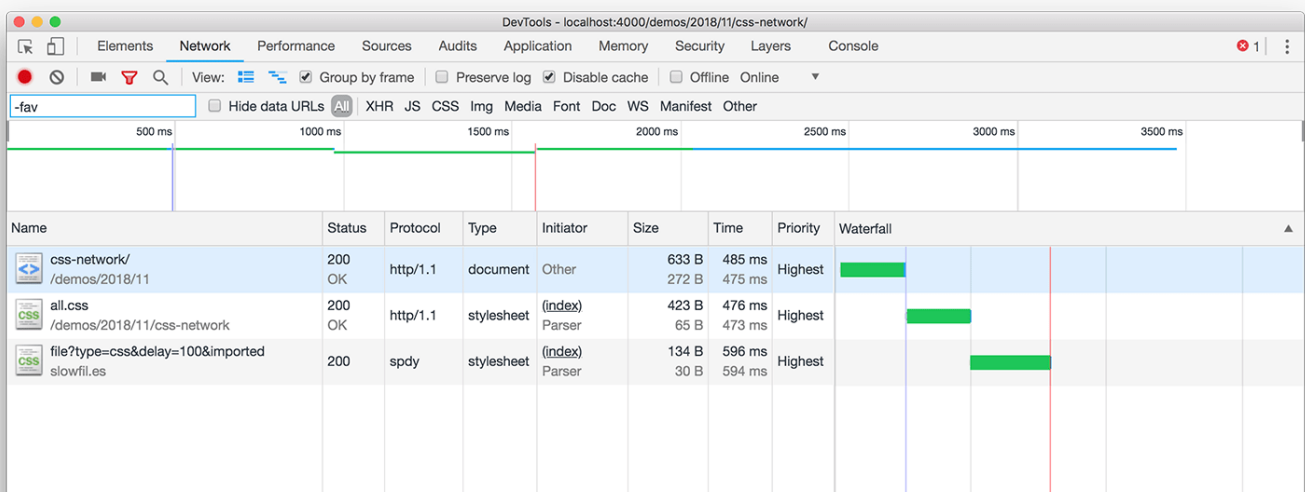
Given the following HTML:

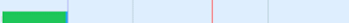

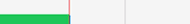
```
<link rel="stylesheet" href="all.css" />
```

...and the contents of *all.css* is:

```
@import url(imported.css);
```

...we end up with a waterfall like this:



Name	Status	Protocol	Type	Initiator	Size	Time	Priority	Waterfall
css-network/ /demos/2018/11	200 OK	http/1.1	document	Other	633 B	485 ms	Highest	
all.css /demos/2018/11/css-network	200 OK	http/1.1	stylesheet	(index) Parser	423 B	476 ms	Highest	
file?type=css&delay=100&imported slowfil.es	200	spdy	stylesheet	(index) Parser	134 B	594 ms	Highest	

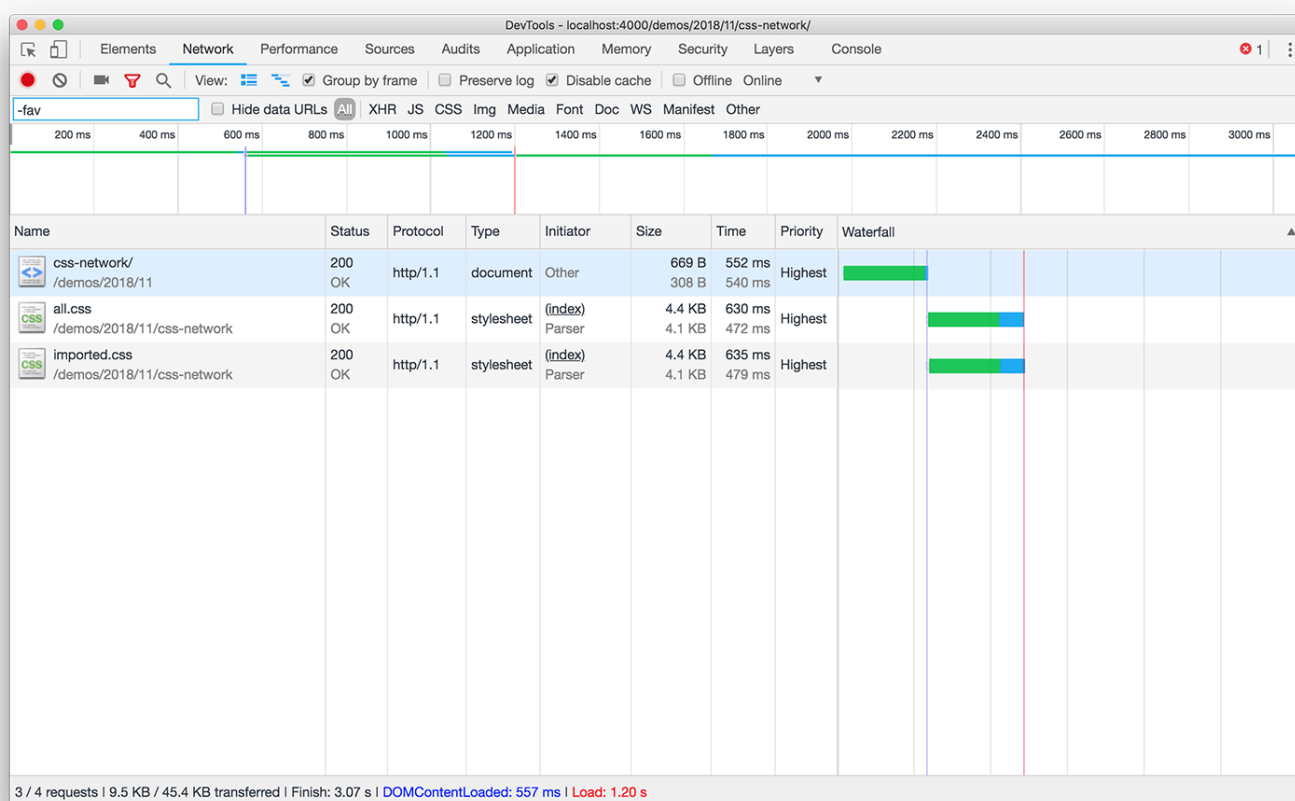
3 / 4 requests | 1.2 KB / 37.0 KB transferred | Finish: 3.47 s | DOMContentLoaded: 490 ms | Load: 1.57 s

Clear lack of parallelisation on the Critical Path.

By simply flattening this out into two `<link rel="stylesheet" />` and zero `@imports`:

```
<link rel="stylesheet" href="all.css" />
<link rel="stylesheet" href="imported.css" />
```

...we get a much healthier waterfall:



Beginning to parallelise our Critical Path CSS.

N.B. I want to briefly discuss an unusual edge case. In the unlikely event that you don't have access to the CSS file that contains the `@import` (meaning you're unable to delete it), you can safely leave it in place in the CSS but *also* complement it with the corresponding `<link rel="stylesheet" />` in your HTML. This means that the browser will initiate the imported CSS' download from the HTML and will skip the `@import`; you won't get any double downloads.

Beware `@import` in HTML

This section is odd. Very odd. I disappeared down such a huge rabbit hole researching this one... Blink and WebKit are broken because of a bug; Firefox and IE/Edge just seem broken. I'm filing the **relevant bugs**.

To fully understand this section we first need to know about the browser's **Preload Scanner**: all major browsers implement a

To fully understand this section we first need to know about the browser's **Preload Scanner**. All major browsers implement a secondary, inert parser commonly referred to as the Preload Scanner. The browser's primary parser is responsible for constructing DOM, CSSOM, running JavaScript, etc., and is constantly stopping and starting as different part of the document block it. The Preload Scanner can safely jump ahead of the primary parser and scan the rest of the HTML to discover references to other subresources (such as CSS files, JS, images). Once they've been discovered, the Preload Scanner begins downloading them ready for the primary parser to pick them up and execute/apply them later. The introduction of the Preload Scanner improved web page performance by around 19%, all without developers having to lift a finger. This is great news for users!

One thing we as developers need to be wary of is inadvertently hiding things from the Preload Scanner, which can happen. More on this later.

This section deals with bugs in WebKit and Blink's Preload Scanner, and an inefficiency in Firefox's and IE/Edge's Preload Scanner.

Firefox and IE/Edge: Place `@import` before JS and CSS in HTML

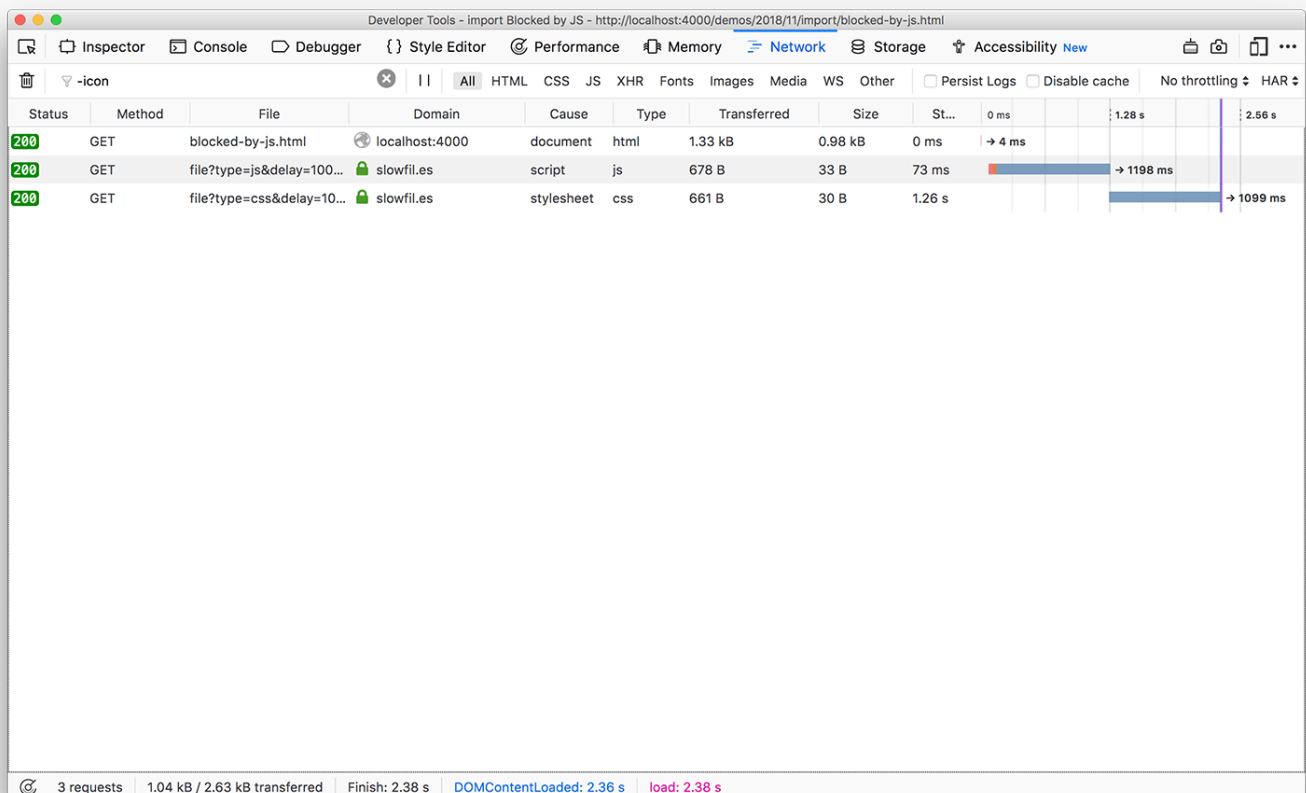
In Firefox and IE/Edge, the Preload Scanner doesn't seem to pick up any `@imports` that are defined after `<script src="">` or `<link rel="stylesheet" />`

That means that this HTML:

```
<script src="app.js"></script>

<style>
  @import url(app.css);
</style>
```

...will yield this waterfall:



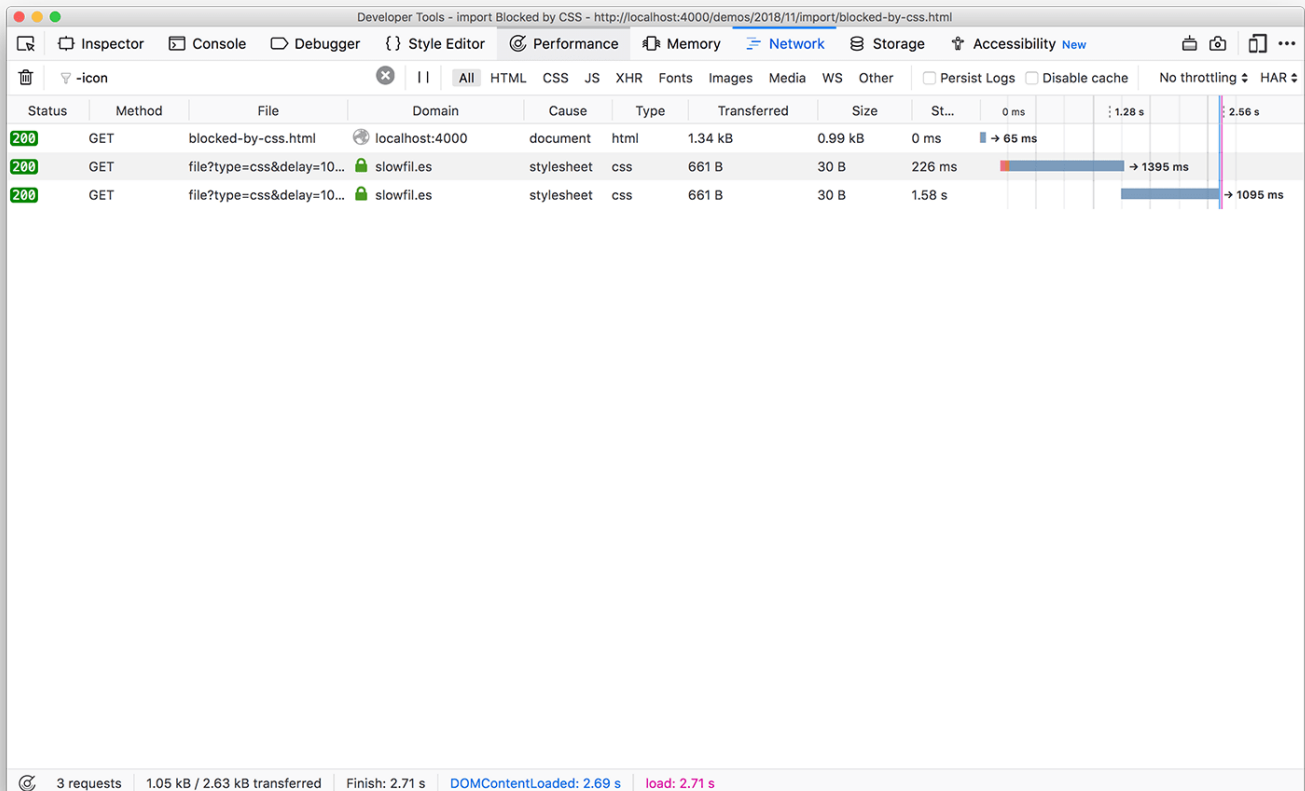
Loss of parallelisation in Firefox due to ineffective Preload Scanner (N.B. The same waterfall occurs in IE/Edge.)

Here we can clearly see that the `@imported` stylesheet does not start downloading until the JavaScript file has completed.

The problem isn't unique to JavaScript, either. The following HTML creates the same phenomenon:

```
<link rel="stylesheet" href="style.css" />

<style>
  @import url(app.css);
</style>
```



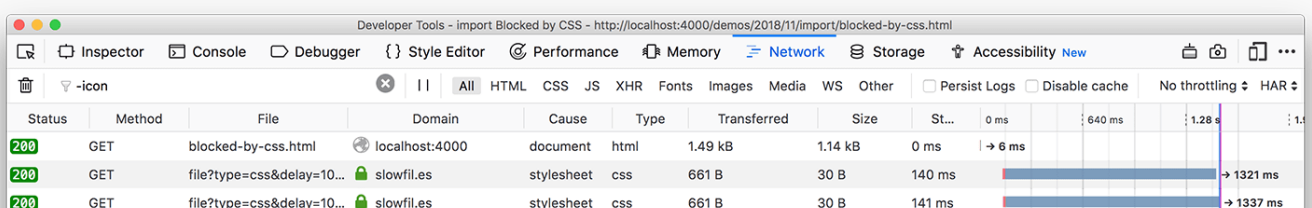
Loss of parallelisation in Firefox due to ineffective Preload Scanner (N.B. The same waterfall occurs in IE/Edge.)

The immediate solution to this problem is to swap the `<script>` or `<link rel="stylesheet" />` and the `<style>` blocks around. However, this will likely break things as we change our dependency order (think *cascade*).

The preferred solution to this problem is to avoid the `@import` altogether and use a second `<link rel="stylesheet" />`:

```
<link rel="stylesheet" href="style.css" />
<link rel="stylesheet" href="app.css" />
```

Much healthier:



3 requests 1.20 kB / 2.78 kB transferred Finish: 1.49 s DOMContentLoaded: 1.48 s load: 1.50 s

Two `<link rel="stylesheet" />` give us back our parallelisation. (N.B. The same waterfall occurs in IE/Edge.)

Blink and WebKit: Wrap `@import` URLs in Quotes in HTML

WebKit and Blink will behave the exact same as Firefox and IE/Edge **only if your `@import` URLs are missing quote marks (")**. This means that the Preload Scanner in WebKit and Blink has a bug.

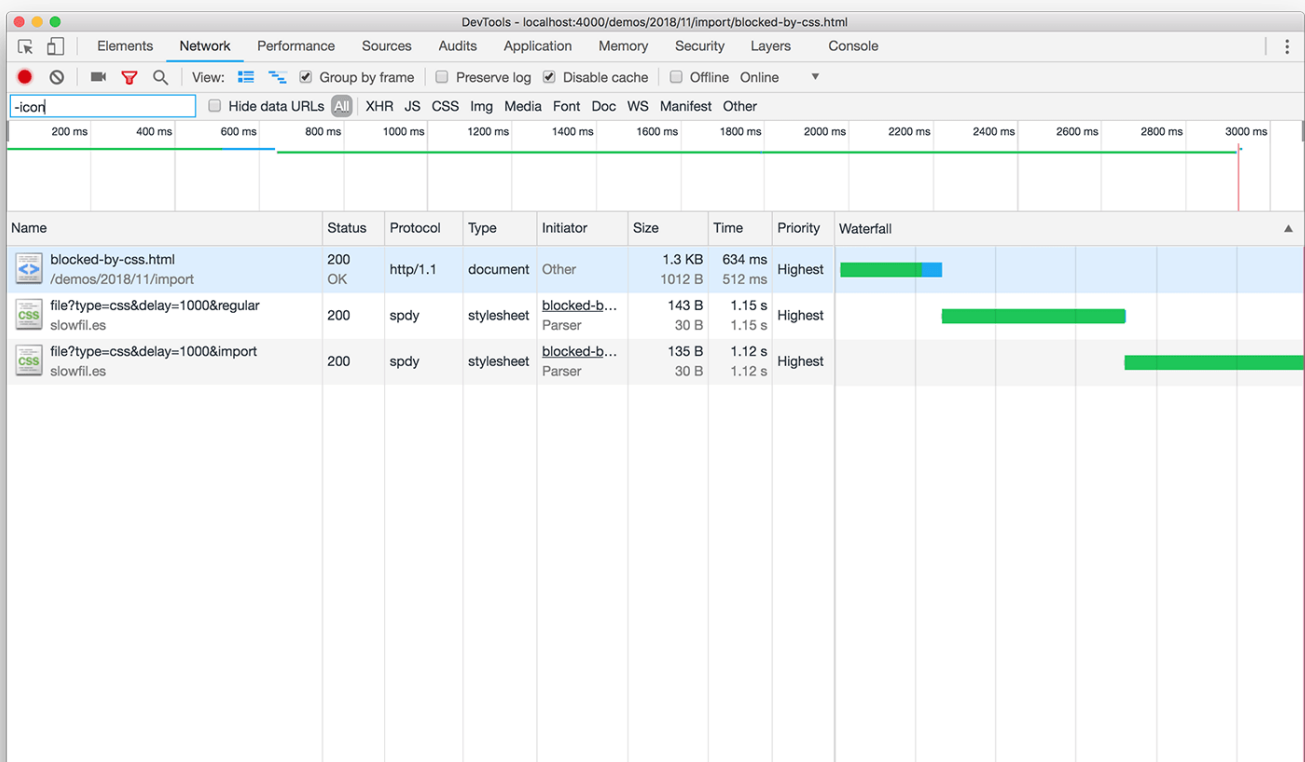
Simply wrapping the `@import` in quotes will fix the problem and you don't need to reorder anything. Still, as before, my recommendation here is to avoid the `@import` entirely and instead opt for a second `<link rel="stylesheet" />`.

Before:

```
<link rel="stylesheet" href="style.css" />
```

```
<style>
  @import url(app.css);
</style>
```

...gives:



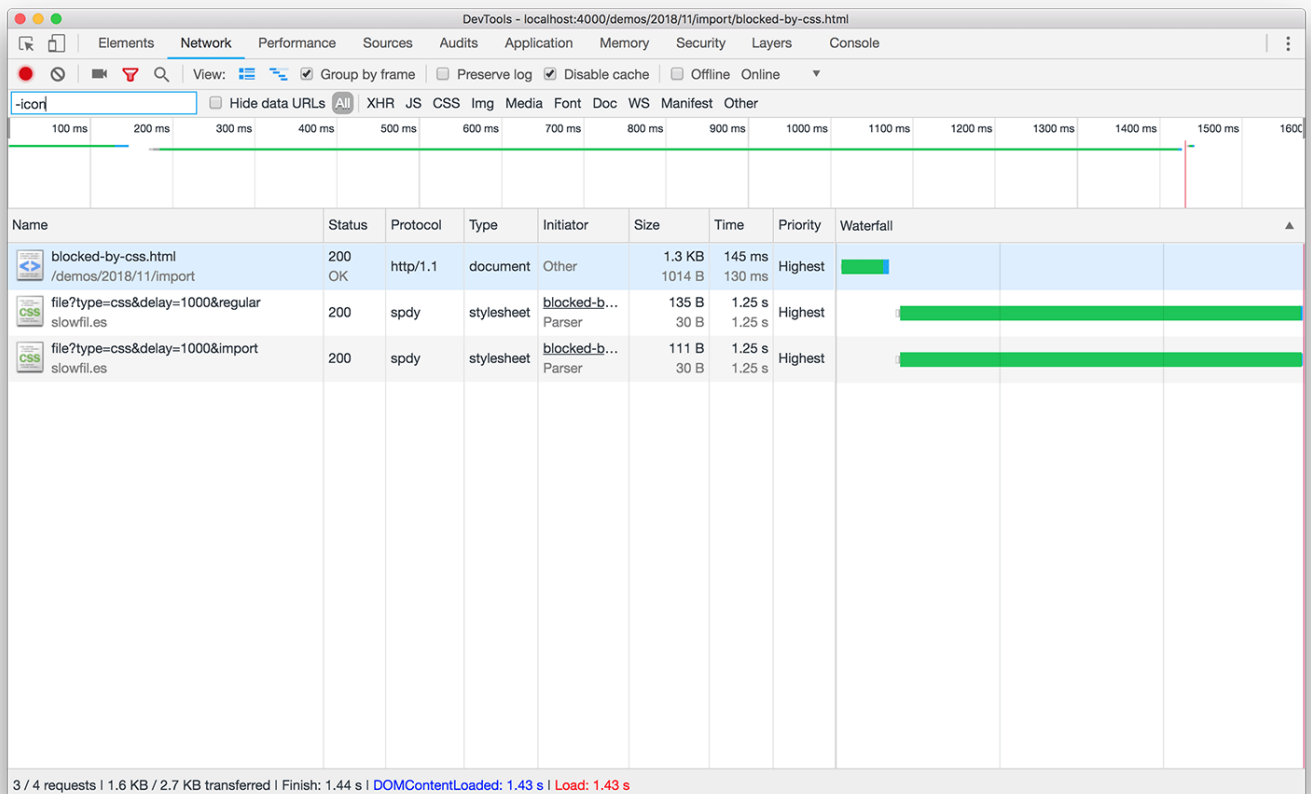
3 / 4 requests | 1.6 KB / 2.7 KB transferred | Finish: 2.93 s | DOMContentLoaded: 2.92 s | Load: 2.92 s

Missing quote marks in our `@import` URLs breaks Chrome's Preload Scanner (N.B. The same waterfall occurs in Opera and Safari.)

After:

```
<link rel="stylesheet" href="style.css" />
```

```
<style>
  @import url("app.css");
</style>
```



Adding quote marks to our `@import` URLs fixes Chrome's Preload Scanner (N.B. The same waterfall occurs in Opera and Safari.)

This is definitely a bug in WebKit/Blink—missing quotes shouldn't hide the `@imported` stylesheet from the Preload Scanner.

Huge thanks to **Yoav** for helping me track this one down.

Yoav has now got a **fix queued up in Chromium**.

Don't Place `<link rel="stylesheet" />` Before Async Snippets

The previous section looked at how CSS can be slowed down by other resources (due to quirks, admittedly), and this section will look at how CSS can inadvertently delay the downloading of subsequent resources, chiefly JavaScript inserted with an asynchronous loading snippet like so:

```
<script>
  var script = document.createElement('script');
  script.src = "analytics.js";
  document.getElementsByTagName('head')[0].appendChild(script);
</script>
```



```
document.getElementsByTagName('head')[0].appendChild(script);
</script>
```

There is a fascinating behaviour present in all browsers that is intentional and expected, yet I have never met a single developer who knew about it. This is doubly surprising when you consider the huge performance impact that it can carry:

A browser will not execute a `<script>` if there is any currently-in flight CSS.

```
<link rel="stylesheet" href="slow-loading-stylesheet.css" />
<script>
  console.log("I will not run until slow-loading-stylesheet.css is downloaded.");
</script>
```

This is by design. This is on purpose. Any synchronous `<script>`s in your HTML will not execute while any CSS is currently being downloaded. This is a simple, defensive strategy to solve the edge case that the `<script>` might ask something about the page's styles: if the script asks about the page's `color` before the CSS has arrived and been parsed, then the answer that the JavaScript gives us could potentially be incorrect or stale. To mitigate this, the browser doesn't execute the `<script>` until the CSSOM is constructed.

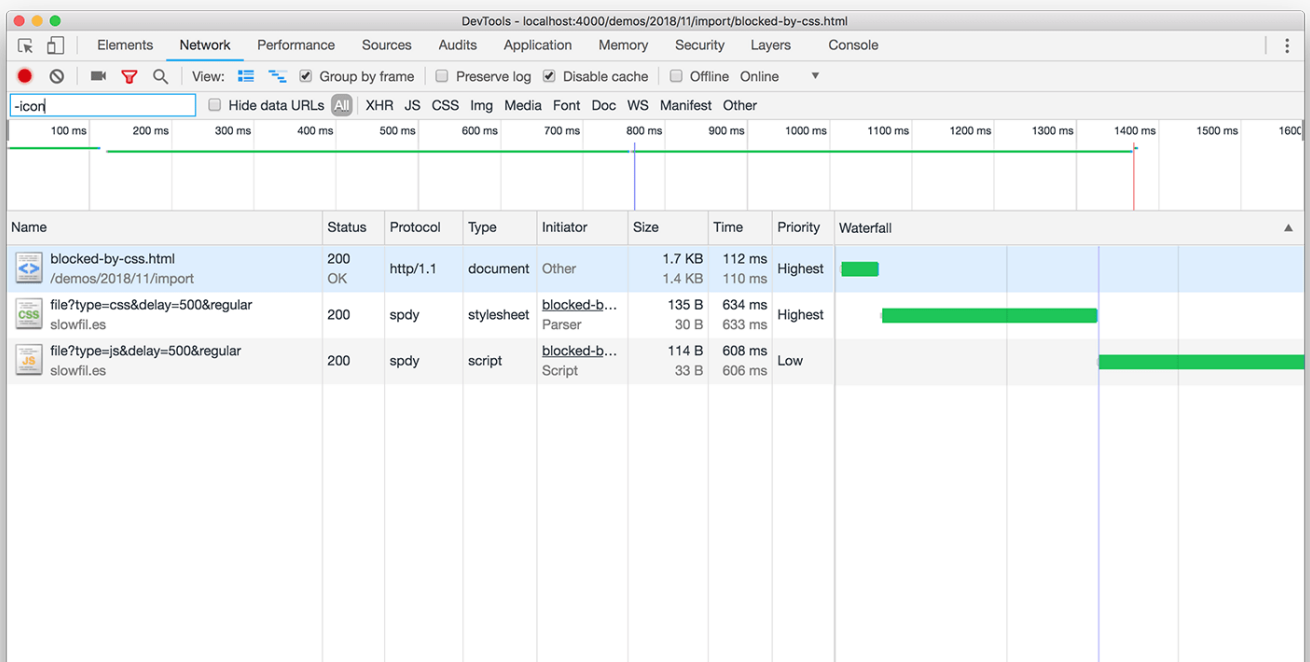
The upshot of this is that any delays on CSS' download time will have a knock-on impact on things like your async snippets. This is best illustrated with an example.

If we drop a `<link rel="stylesheet" />` in front of our async snippet, it will not run until that CSS file has been downloaded and parsed. This means our CSS is pushing everything back:

```
<link rel="stylesheet" href="app.css" />

<script>
  var script = document.createElement('script');
  script.src = "analytics.js";
  document.getElementsByTagName('head')[0].appendChild(script);
</script>
```

Given this order, we can clearly see that the JavaScript file does not even begin downloading until the moment the CSSOM is constructed. We've completely lost any parallelisation:



3 / 4 requests | 1.9 KB / 3.1 KB transferred | Finish: 1.37 s | DOMContentLoaded: 761 ms | Load: 1.37 s

Having a stylesheet before an async snippet undoes our opportunity to parallelise.

Interestingly, the Preload Scanner would like to have picked up the reference to `analytics.js` ahead of time, but we've inadvertently hidden it: `"analytics.js"` is a string, and doesn't become a tokenisable `src` attribute until the `<script>` element exists in the DOM. This is the bit I meant earlier when I said 'More on this later.'

It's very common for third party vendors to provide async snippets like this to more safely load their scripts. It's also very common for developers to be suspicious of these third parties and place their async snippets later in the page. While this is done with the best of intentions—'I don't want to put third party `<script>`s before my own assets!'—it can often be a net loss. In fact, Google Analytics even tell us what to do, and they're right:

"Copy and paste this code as the first item into the `<HEAD>` of every webpage you want to track."

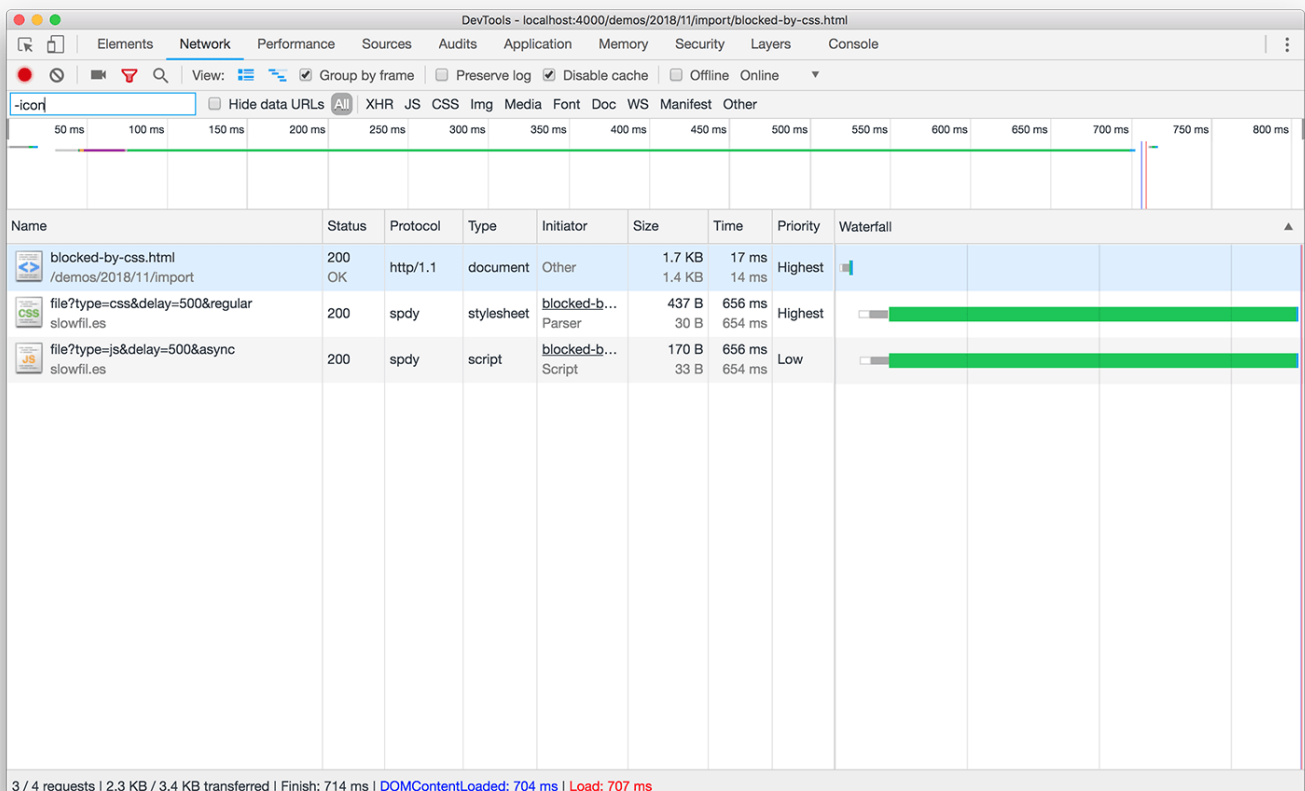
So my advice here is:

If your `<script>...</script>` blocks have no dependency on CSS, place them above your stylesheets.

Here's what happens when we move to this pattern:

```
<script>
  var script = document.createElement('script');
  script.src = "analytics.js";
  document.getElementsByTagName('head')[0].appendChild(script);
</script>

<link rel="stylesheet" href="app.css" />
```



Swapping a stylesheet and an async snippet around can regain parallelisation.

Now you can see that we've completely regained parallelisation and the page has loaded almost 2× faster.

Place Any Non-CSSOM Querying JavaScript Before CSS; Place Any CSSOM-Querying JavaScript After CSS

Dang. This article is getting way, way more forensic than I intended.

Taking this even further, and looking beyond just async loading snippets, how should we load CSS and JavaScript more generally? To work that out, I posed myself the following question and worked back from there:

If

- synchronous JS defined after CSS is blocked on CSSOM construction, and;
- synchronous JS blocks DOM construction...

then—assuming no interdependencies—which is faster/preferred?

- Script then style;
- style then script?

The answer:

If the files do not depend on one another, then you should place your blocking scripts above your blocking styles—there's no point delaying the JavaScript execution with CSS upon which the JavaScript doesn't actually depend.

(The Preload Scanner ensures that, even though DOM construction is blocked on the scripts, the CSS is still downloaded in parallel.)

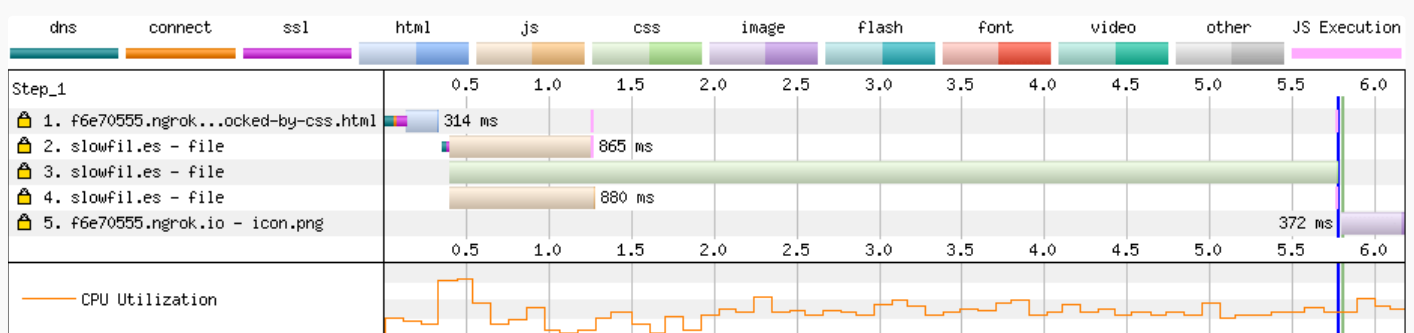
If some of your JavaScript does but some does not depend on CSS, then the absolute most optimum order for loading synchronous JavaScript and CSS would be to split that JavaScript in two and load it either side of your CSS:

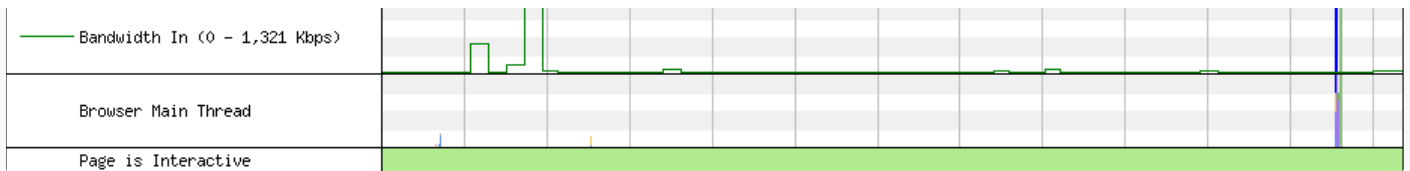
```
<!-- This JavaScript executes as soon as it has arrived. -->
<script src="i-need-to-block-dom-but-DONT-need-to-query-cssom.js"></script>

<link rel="stylesheet" href="app.css" />

<!-- This JavaScript executes as soon as the CSSOM is built. -->
<script src="i-need-to-block-dom-but-DO-need-to-query-cssom.js"></script>
```

With this loading pattern, we get download and execution both happening in the most optimum order. I apologise for the tiny, tiny details in the below screenshot, but hopefully you can see the small pink marks that represent JavaScript execution. Entry (1) is the HTML that is scheduled to execute some JavaScript when other files arrive and/or execute; entry (2) executes the moment it arrives; entry (3) is CSS, so executes no JavaScript whatsoever; entry (4) doesn't actually execute until the CSS is finished.





How CSS can impact the points at which JavaScript executes.

N.B. It is imperative that you test this pattern against your own specific use-case: there could be different results depending on whether or not there are large differences in file-size and execution costs between your before-CSS JavaScript file and the CSS itself. Test, test, test.

Place `<link rel="stylesheet" />` in `<body>`

This final strategy is a relatively new one, and has great benefit for perceived performance and progressive render. It's also very component friendly.

In HTTP/1.1, it's typical that we concatenate all of our styles into one main bundle. Let's call that `app.css`:

```
<!DOCTYPE html>
<html>
<head>

  <link rel="stylesheet" href="app.css" />

</head>
<body>

  <header class="site-header">

    <nav class="site-nav">...</nav>

  </header>

  <main class="content">

    <section class="content-primary">

      <h1>...</h1>

      <div class="date-picker">...</div>

    </section>

    <aside class="content-secondary">

      <div class="ads">...</div>

    </aside>

  </main>

  <footer class="site-footer">
</footer>

</body>
```

This carries three key inefficiencies:

1. **Any given page will only use a small subset of styles found in `app.css`:** we're almost definitely downloading more CSS than we need.
2. **We're bound to an inefficient caching strategy:** a change to, say, the background colour of the currently-selected day on a date picker used on only one page, would require that we cache-bust the entirety of `app.css`.
3. **The whole of `app.css` blocks rendering:** it doesn't matter if the current page only needs 17% of `app.css`, we still have to wait for the other 83% to arrive before we can begin rendering.

With HTTP/2, we can begin to address points (1) and (2):

```
<!DOCTYPE html>
<html>
<head>

  <link rel="stylesheet" href="core.css" />
  <link rel="stylesheet" href="site-header.css" />
  <link rel="stylesheet" href="site-nav.css" />
  <link rel="stylesheet" href="content.css" />
  <link rel="stylesheet" href="content-primary.css" />
  <link rel="stylesheet" href="date-picker.css" />
  <link rel="stylesheet" href="content-secondary.css" />
  <link rel="stylesheet" href="ads.css" />
  <link rel="stylesheet" href="site-footer.css" />

</head>
<body>

  <header class="site-header">

    <nav class="site-nav">...</nav>

  </header>

  <main class="content">

    <section class="content-primary">

      <h1>...</h1>

      <div class="date-picker">...</div>

    </section>

    <aside class="content-secondary">

      <div class="ads">...</div>

    </aside>

  </main>

  <footer class="site-footer">
</footer>

</body>
```

</body>

Now we're getting some way around the redundancy issue as we're able to load CSS more appropriate to the page, as opposed to indiscriminately downloading everything. This reduces the size of the blocking CSS on the Critical Path.

We're also able to adopt a more deliberate caching strategy, only cache busting the files that need it and leaving the rest untouched.

What we haven't solved is the fact that it all still blocks rendering—we're still only as fast as our slowest stylesheet. What this means is that if, for whatever reason, `site-footer.css` takes a long time to download, the browser can't make a start on rendering `.site-header`.

However, due to a recent change in Chrome (version 69, I believe), and behaviour already present in Firefox and IE/Edge, `<link rel="stylesheet" />`s will only block the rendering of subsequent content, rather than the whole page. This means that we're now able to construct our pages like this:

```
<!DOCTYPE html>
<html>
<head>

  <link rel="stylesheet" href="core.css" />

</head>
<body>

  <link rel="stylesheet" href="site-header.css" />
  <header class="site-header">

    <link rel="stylesheet" href="site-nav.css" />
    <nav class="site-nav">...</nav>

  </header>

  <link rel="stylesheet" href="content.css" />
  <main class="content">

    <link rel="stylesheet" href="content-primary.css" />
    <section class="content-primary">

      <h1>...</h1>

      <link rel="stylesheet" href="date-picker.css" />
      <div class="date-picker">...</div>

    </section>

    <link rel="stylesheet" href="content-secondary.css" />
    <aside class="content-secondary">

      <link rel="stylesheet" href="ads.css" />
      <div class="ads">...</div>

    </aside>

  </main>
```

```
<link rel="stylesheet" href="site-footer.css" />
<footer class="site-footer">
</footer>

</body>
```

The practical upshot of this is that we're now able to progressively render our pages, effectively drip-feeding styles to the page as they become available.

In browsers that don't currently support this new behaviour, we suffer no performance degradation: we fall back to the old behaviour where we're only as fast as the slowest CSS file.

For further detail on this method of linking CSS, I would recommend reading [Jake's article on the subject](#).

Summary

There is a *lot* to digest in this article. It ended up going way beyond the post I initially intended to write. To attempt to summarise the best network performance practices for loading CSS:

- Lazyload any CSS not needed for Start Render:
 - This could be Critical CSS;
 - or splitting your CSS into Media Queries.
- Avoid `@import`:
 - In your HTML;
 - but in CSS especially;
 - and beware of oddities with the Preload Scanner.
- Be wary of synchronous CSS and JavaScript order:
 - JavaScript defined after CSS won't run until CSSOM is completed;
 - so if your JavaScript doesn't depend on your CSS;
 - load it before your CSS;
 - but if it does depend on your CSS:
 - load it after your CSS.
- Load CSS as the DOM needs it:
 - This unblocks Start Render and allows progressive rendering.

Warning

Everything I have outlined above adheres to specs or known/expected behaviour, but, as always, test everything yourself. While it's all theoretically true, things always work differently in practice. Test and **measure**.

Thanks

I'm grateful to **Yoav**, **Andy**, and **Ryan** for their insights and proof-reading over the last couple of days.

Did you enjoy this? Hire me!

Hi there, I'm Harry. I am an **award-winning Consultant Front-end Architect, designer, developer, writer and speaker** from the UK. I **write, tweet, speak** and **share code** about authoring and scaling CSS for big websites. You can **hire me**.

Follow [@csswizardry](#)

I am currently **accepting new projects** for Q1–2 2019



The new generation of project management tools is here and it's visual.

ADS VIA CARBON

Projects



inuitcss

ITCSS – coming soon...

css { guide: lines; }

Next Appearance

Workshop

🚩 **New Adventures**, Nottingham (England), January 2019

Newsletter

Infrequent updates, special offers, and exclusive content. [Learn more...](#)

Email Address

email@domain.com

Join

I am available for hire to consult, advise, and develop with passionate product teams across the globe.

I specialise in large, product-based projects where performance, scalability, and maintainability are paramount.

I am currently considering new projects for Q1–2 2019.

CSS Wizardry Ltd is a company registered in England and Wales. **Company No.** 08698093, **VAT No.** 170659396