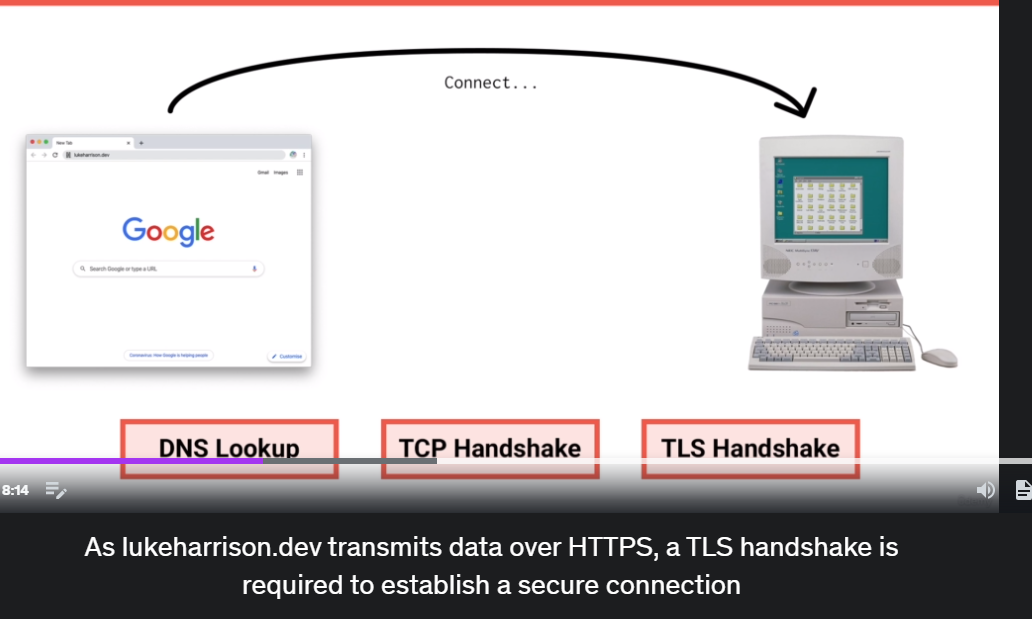
Perceived performance



**DNS lookups** happen for each unique domain and have a cost.

If you have assets such as CSS or javascript stored on a CDN, each of these domain names will require their own DNS lookup as well.

**TCP Handshake**

Now that the browser the IP address of the server where the website’s files are stored, it can connect.

This is done through what’s called a TCP Handshake, which negotiates a connection between the user’s browser and the website’s server so that data can be sent and received freely.

**TLS Handshake**

It’s requried to establish a secure connection

**Round trip**

Any journey to the server and then back again with a response is called a round trip.

8. How Browsers Load websites

Once HTML is fully passed and the DOM is complete, the browser fires DOMContentLoaded event.

9. Measuring Web Performance

Loading Performance , Rendering Performance

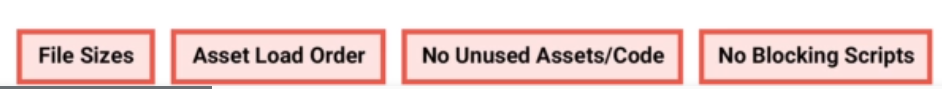
Loading Performance is the area of work performance optimization, focusing around optimizaing the delivery of assets from a server to the browser.

Are file sizes low enough?

Does the browser receive critical assets first?

Do we only load what we need to load?

Is Javascript execution unnecessarily blocking render?



Rendering performance focuses on optimizing how the browser uses the downloaded assets to render the website and handle any subsequent interactions.

How complex are CSS style calcuations?

Is Javascript causing unnecessary reflows?

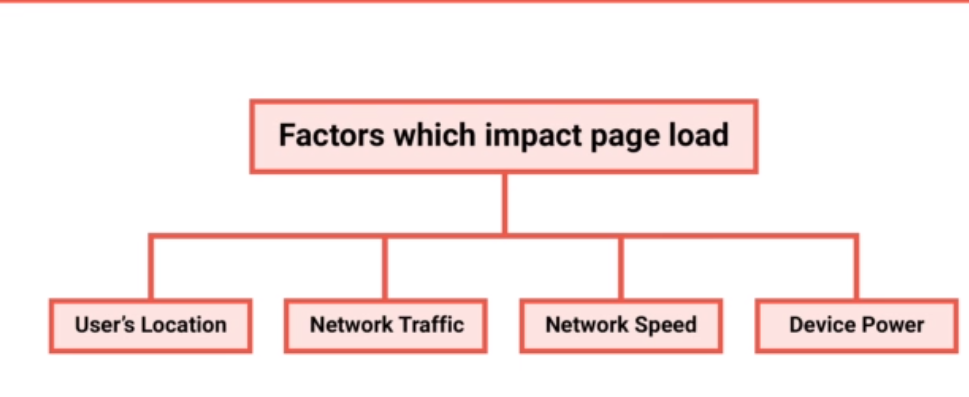
Are my event listeners making the page choppy?

When measuring performance, there’s two main types of data sets which you can draw upon.

Lab Data , Field Data

Lab Data is performance data captured using synthetic testing like google page speed

Field data refers to performance data which is collected from real life users visiting a website out there on the World Wide Web. Like google analytics

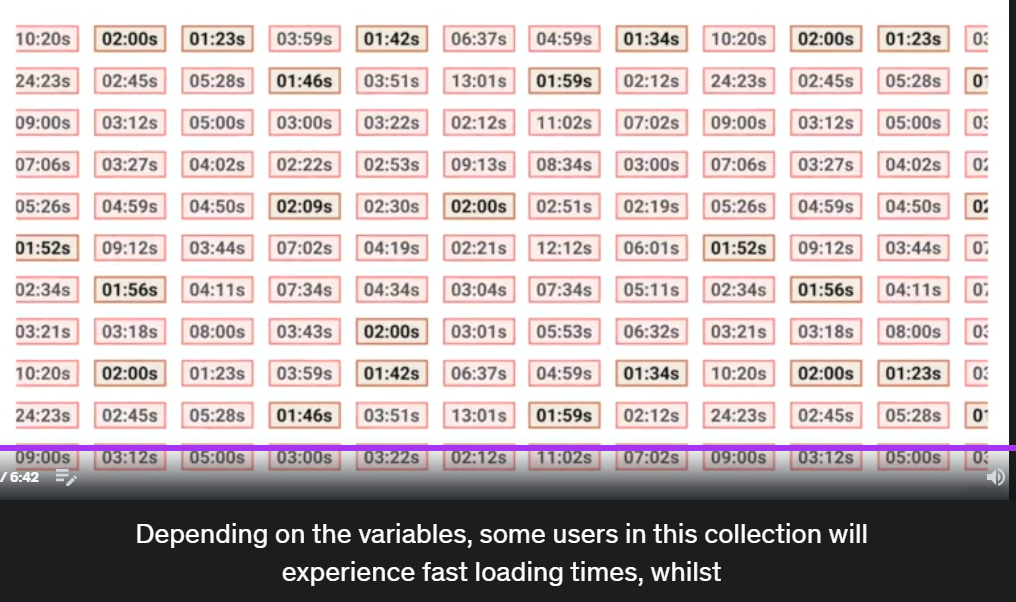


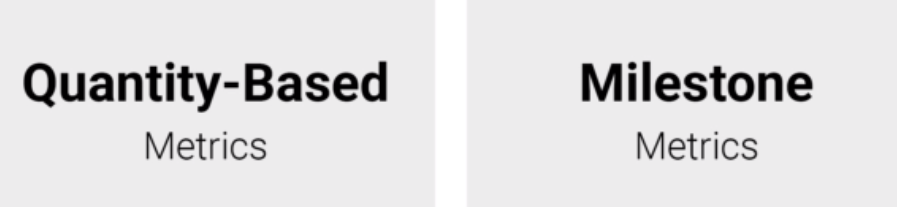
Users Location : impacts connection time

Network traffic: server response times

Network speed: how fast assets can be downloaded

Device Poser: how quickly they can be then processed,





Quantity based metrics measure things such as the number of HTTP requests, the size of the page in bytes, the number of images loaded, basically, anything you can count.

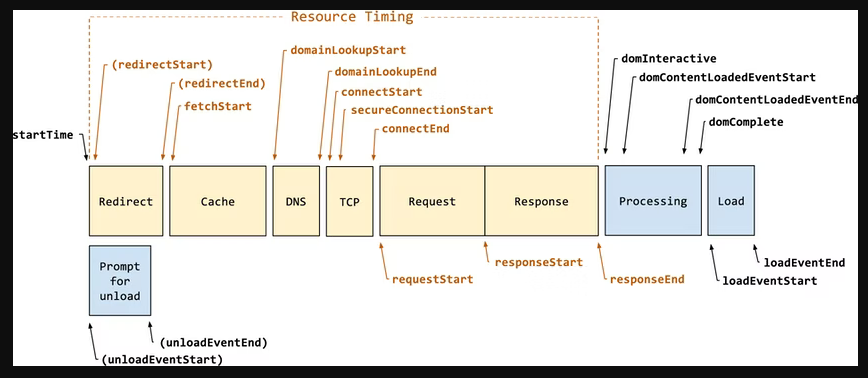
Miestone metrics measure how long it takes the browser to reach a specific phase of the loading process.

How long the server takes to respond, how long it takes the browser to start rendering content or how long it takes for that content to become interactive once Javascript execution is completed.

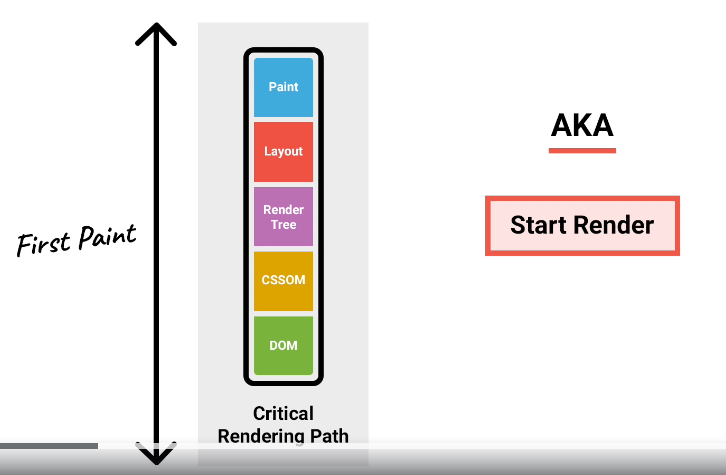
Whilst quantity bases metrics are relatively self-explanatory, milestone metrics require a little more elaboration.

What is TTFB? [#](https://web.dev/ttfb/#what-is-ttfb) https://web.dev/ttfb/

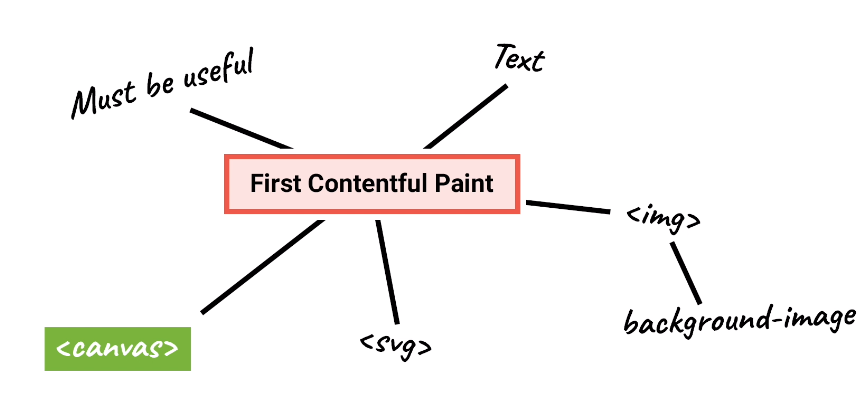
TTFB is a metric that measures the time between the request for a resource and when the first byte of a response begins to arrive.



First Paint also known as “Start Render”



First Contentful Paint, which is very similar metric like First Paint, it measures when the first bits of content are rendered to the screen, but makes a distinction that the content must be useful to the end user.



**Speed Index**

In the past, it has been common to track a website’s speed y measuring when the browser fires events such as DOMContentLoaded and Load.

Speed index tries to fix this inaccuracy by measuring the average time it takes for only the viewport content to be rendered, with any below the fold elements now safely ignored.

This also means that when comparing page loads between mobile and desktop devices, the mobile devices will always have a lower Speed index simply because there’s just less viewport content to render.

**Largest Contentful Paint.**

Goals are similar to Speed Index, identify when the main content of the page has loaded.

Largest Contentful Paint on the other hand, purely looks at when the largest images or text block is visible within the viewport, which would itself be indicative of everything else being visible in the viewport as well.

**Time to Interactive**

It measures the point in the loading process where the page is fully interactive.

For this to happen, the page must be complete enough to facilitate interaction, which makes sense as you can’t interact with what isn’t there. This means the page must be displaying useful content, which we’re tracking using First Contentful Paint.

Event handlers must be registered to this content and then also respond to user input consistently within 50 milliseconds.

**Time to First CPU Idle**, sometimes called the First Interactive

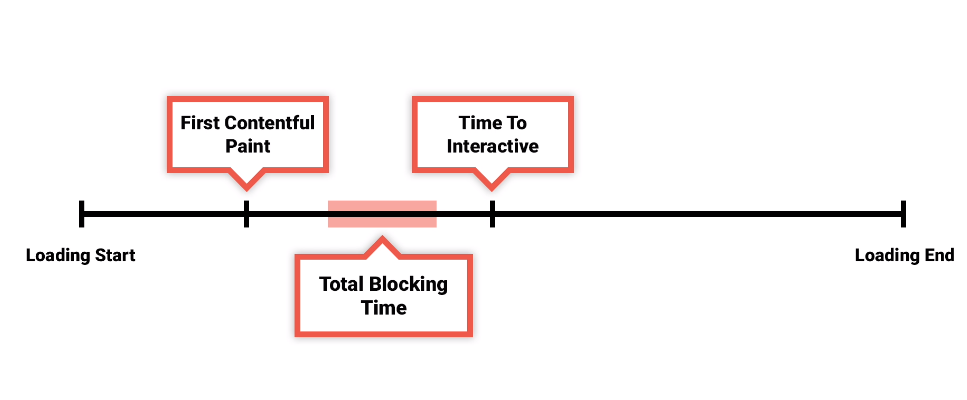
It measures when the page is first expected to be interactive in some capacity and responding to user input quickly.

So to reiterate this, ***Time to Interactive*** measures when the page is fully interactie, whilst ***Time to First CPU Idle*** measures when the page is first interactive.

**Total Blocking Time**

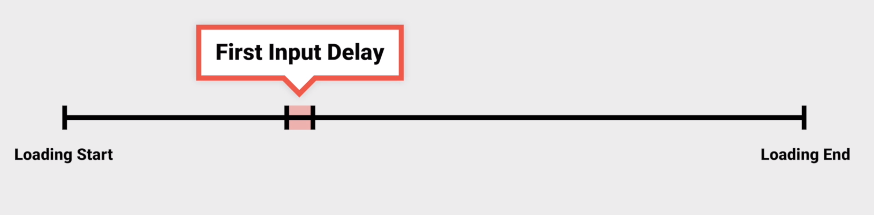
It’s another JavaScript oriented performance metric, it measures the amount of time between ***First Contentful Paint*** and at ***Time to Interactive*** where the browser was too busy executing JavaScript to accept input, which the user will perceive as slow or janky.

It makes a good complementary metric to ***Time to Interactive***, as it can help developers understand just how non interactive a page is prior to the Time to Interactive metric point.



**First Input Delay or Input Latency**

It represents the time from when the user is first able to interact with elements on the page to the point where these elements are then able to respond to those interactions, which, depending on what the browser is doing, isn’t always immediately. This is because browsers are single threaded, which in layman’s terms means they can only execute instructions in a single sequence, one at a time.



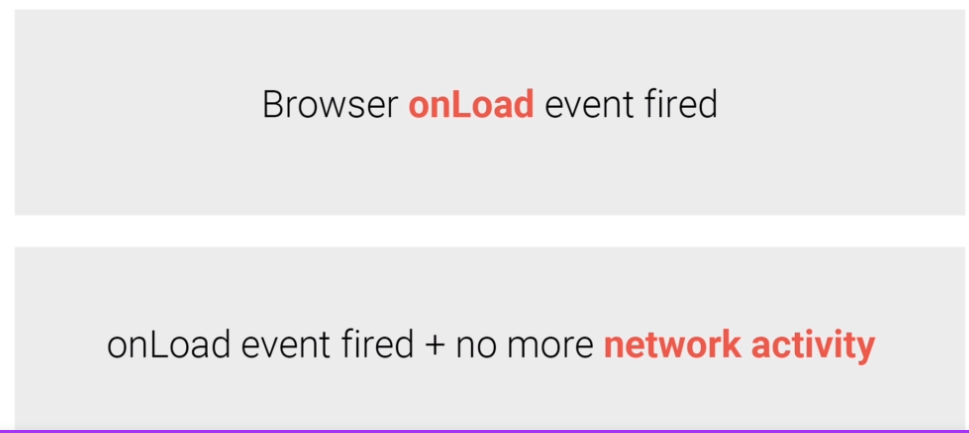
So with this mind, if the page content has finished rendering, but the browser is still busy executing JavaScript, any user input during this time will not be registered, as the browser doesn’t have the resources available to respond.

A poor First Input Delay usually equals user frustration because interactions don’t appear to be snappy or responsive, which nobody likes.

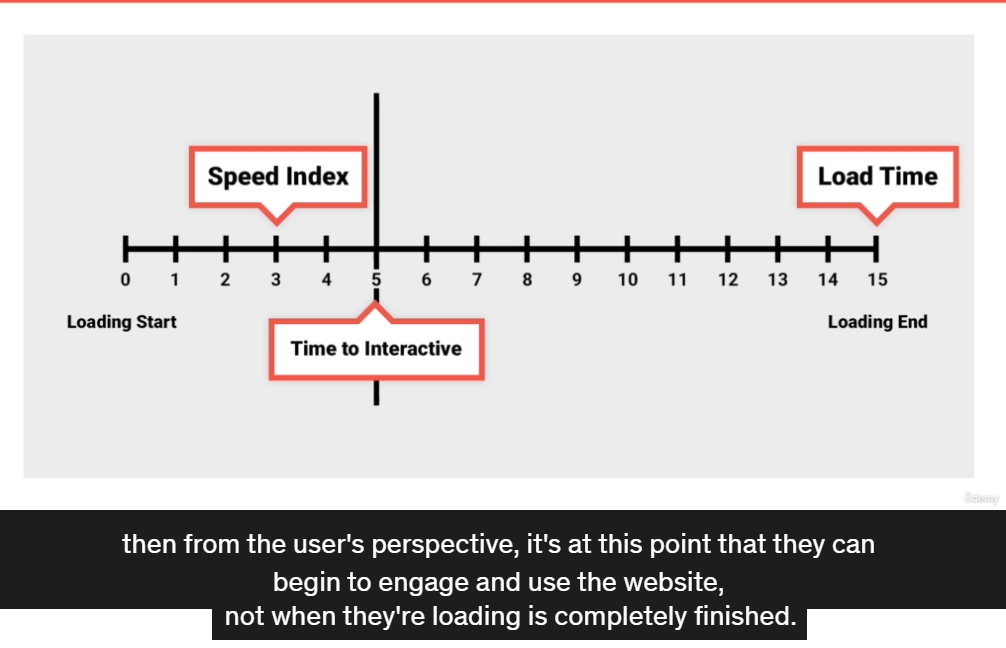
**Load Time**

Depending on the tool, load time can represent a few different points in the loading process.

Where the browser onload event is fired and all of the page static content has loaded, or when the above has happened, but there’s been no additional network activity for a few seconds, signaling that it’s come to end.



As we’re already covered, this metric is less important because it’s not an accurate representation of the user’s own experience of loading.



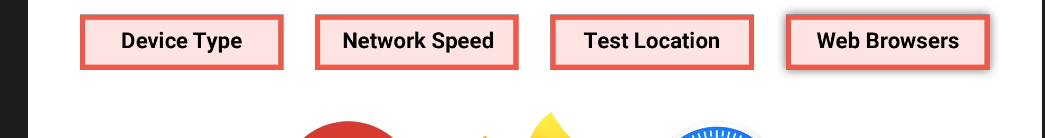
**Cumulative Layout Shift**

Often, as the website is loading, during the render phase, the layout of a page can suddenly shift as new elements are introduced.

This can be very frustrating for a user who’s trying to engage or interact with the page.

It measures these unstable elements and how often they trigger unnecessary reflows.

The lower the score, the better.



8-25 HTTP Requests

Web Page Test

Advanced Configuration > Chromium > Command-line

--disable-http2 –disable-guid

The first disables HTTP/2, whilst the second disables any HTTP/3 implementations.

8-27 Image Resizing

<picture>

<source srcset=”img/b.jpg” type=”image/jpg” media=”(max-width: 768px)”/>

<source srcset=”img/c.jpg” type=”image/jpg” media=”(max-width: 400px)”/>

<img src=”img/a.jpg” alt=”MLE Power”/>

</picture>

8-28 Image optimizing

Using imageoptim, a free app on Mac, we can remove unnecessary data within images such as metadata, thumbnails and unnecessary color profiles which aren’t needed on the web.

However, this app isn’t avaialbe on windows, and using it means we lose our raw on unoptimized images.

Instead, let’s automate this optimization using our existing gulp build process.

Baseline JPG, Progressive JPG

WebP images don’t support progressive rendering, so a local baseline JPGs.

This means for key viewport images like this carousel slide, you may want to see the original JPG instead to prioritize perceived performance.

Though it’s important to test both, as the difference in the WebP file size may offset any advantage a progressive JPG brings to render speed.

Here, Even on 2G, The difference is negligible.

<video autoplay loop muted playsinline>

<source src=”/img/dog.webm” type=”video/webm”>

<source src=”/img/dog.mp4” type=”video/mp4”>

</video>

By placing the WebM source first, if the browser doesn’t support that format, then it falls back to the MP4.

8-29 Resource Hints

Resource Hints are a way we can improve web performance by telling the browser that it needs to make connections to other domains or even download content before it normally would.

Hence the name resource hints, as you are giving the browser a hot tip on what’s to come.

You create resource hints in the head of a HTML document using a link element like this.

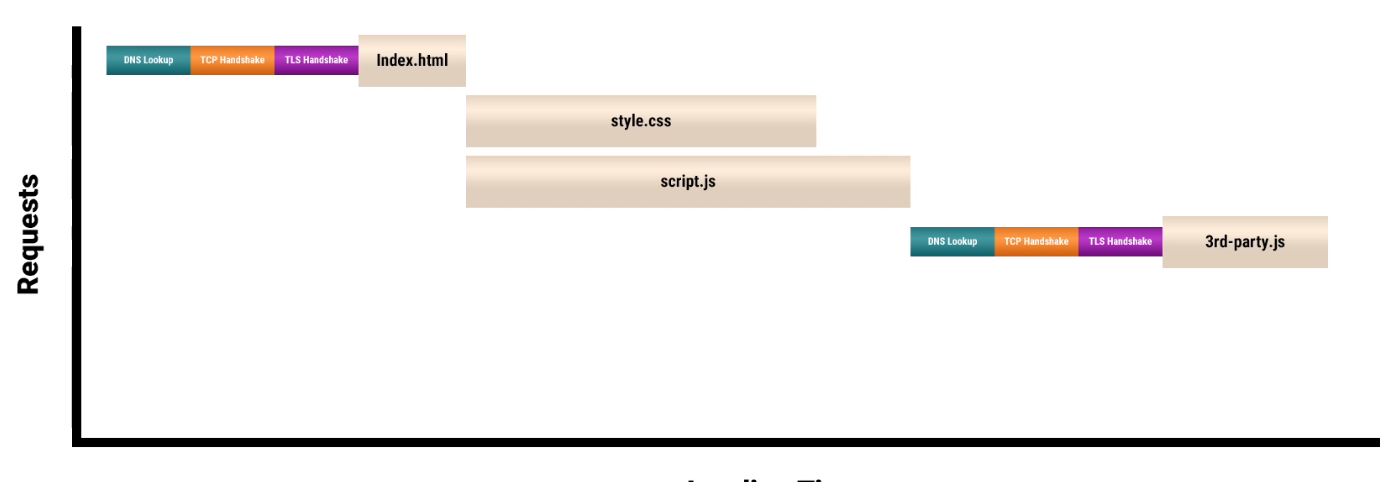
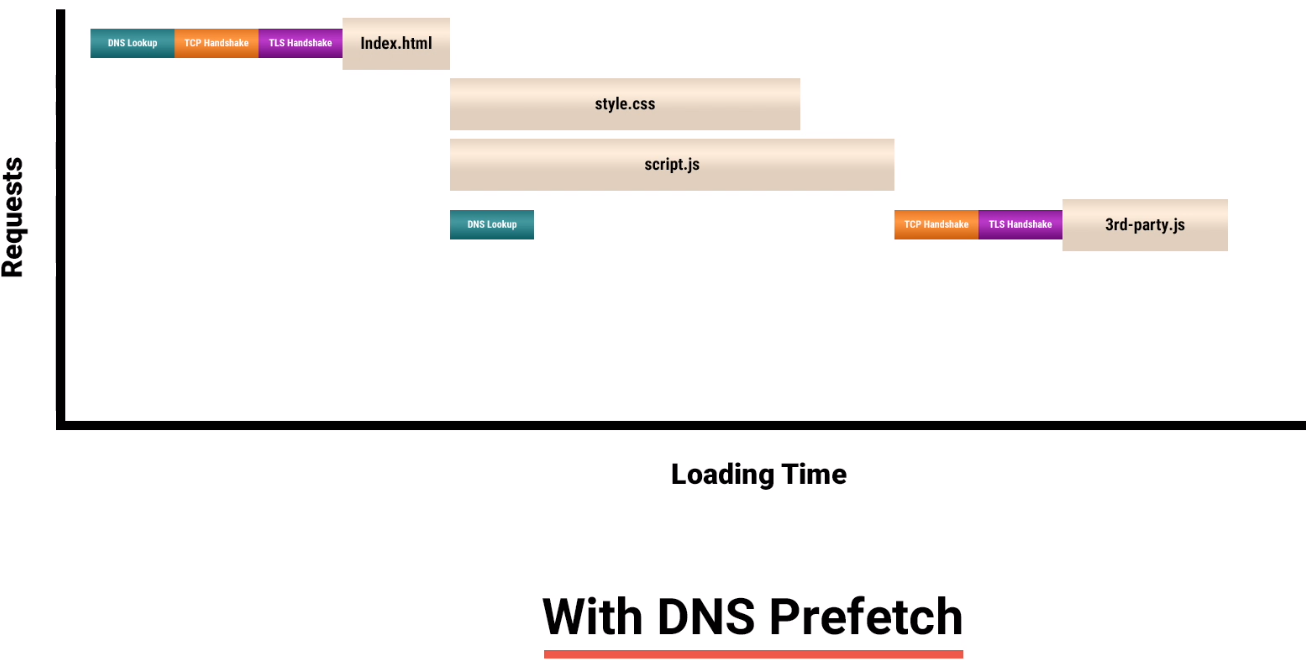


There are a number of different types of resource hints.

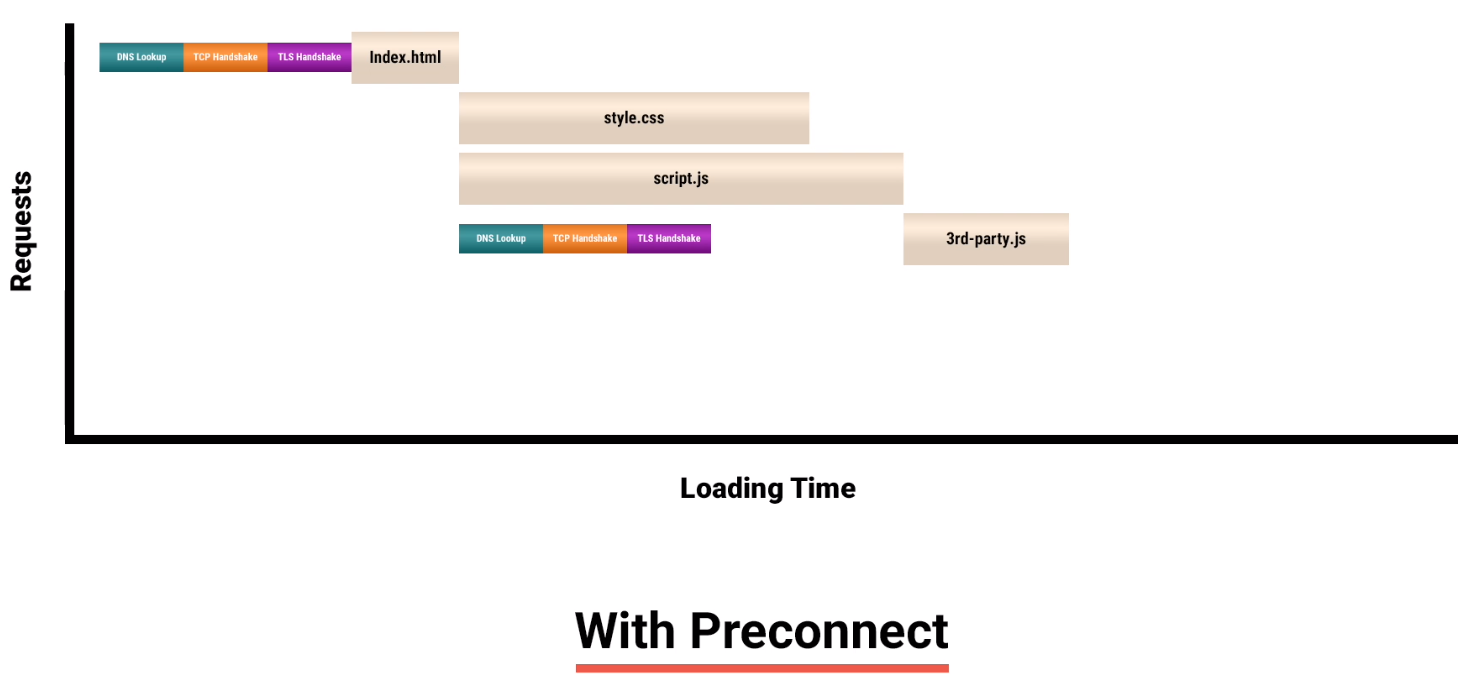
DNS Prefetch



This tells the browser to perform a DNS look up on a domain, which, as we’ve learned, is one of the steps required for the browser to connect to the server.

So by the time the browser has discovered it needs to download a resource from the domain, it’s able to form a connection a little bit quicker than otherwise because part of the work is already done.  

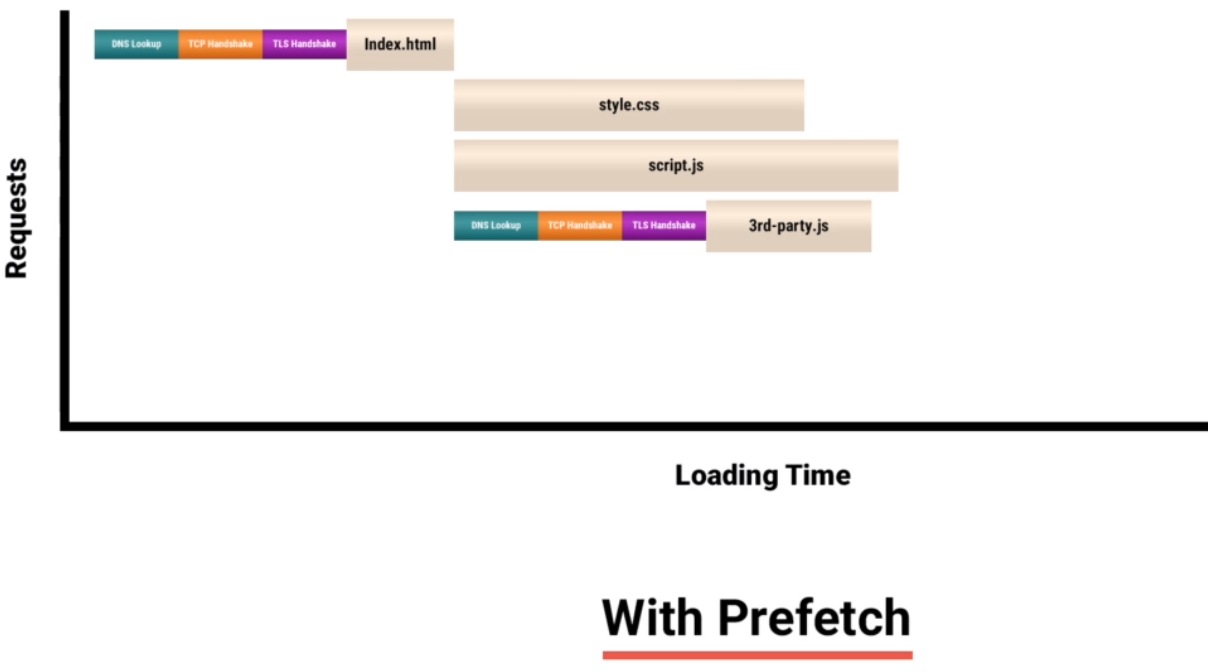
Preconnect it’s a lot like DNS Prefetch, except that instead of just performing a DNS lookup early, the browser then goes on to make the TCP handshake and optional TLS handshake if HTTPS. This allows the browser to ready a full connection early, so when it later discovers that it needs to download a resource, a connection is already open and ready to go.



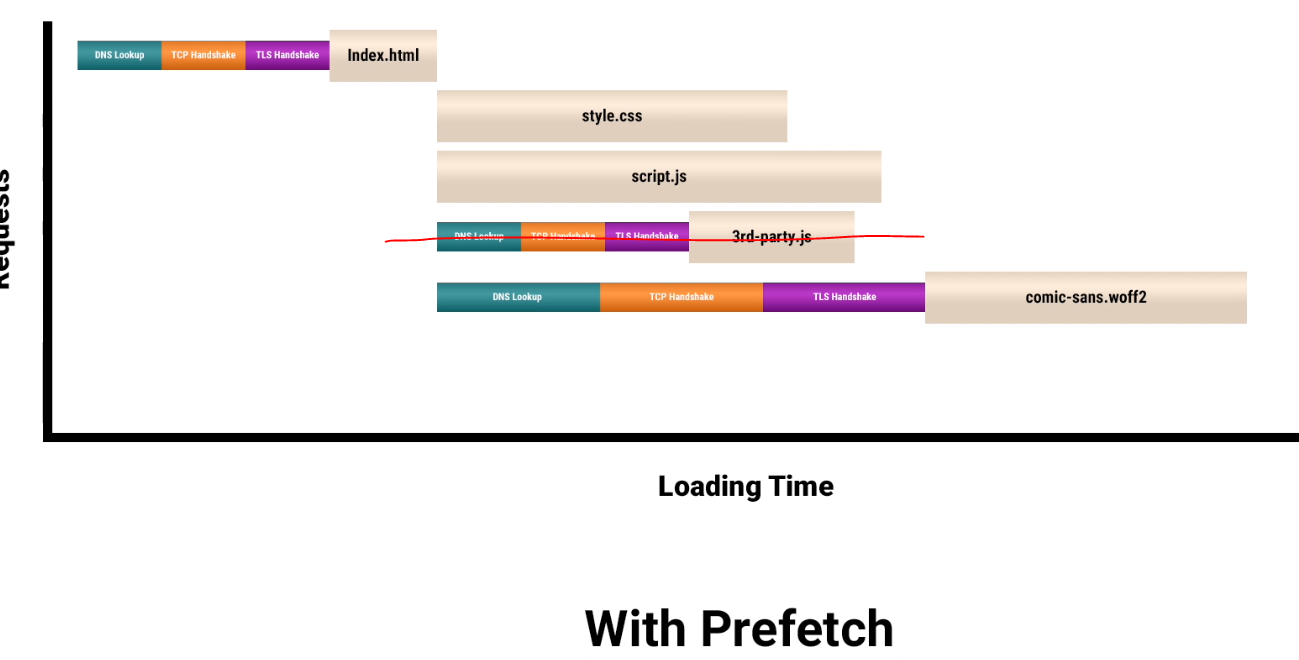


If we know we’ll definitely need a resource later in the loading process, we can tell the browser using a Prefetch resource hint that it should download and cache the file in preparation.

This could be css, javascript or even a HTML document, anything that cachable really.



However this resource hint isn’t definitive. If the browser determines it has more important resources to load, like a large font file on a slow network, for example, it may abandon the Prefetch to focus on what it deems important.

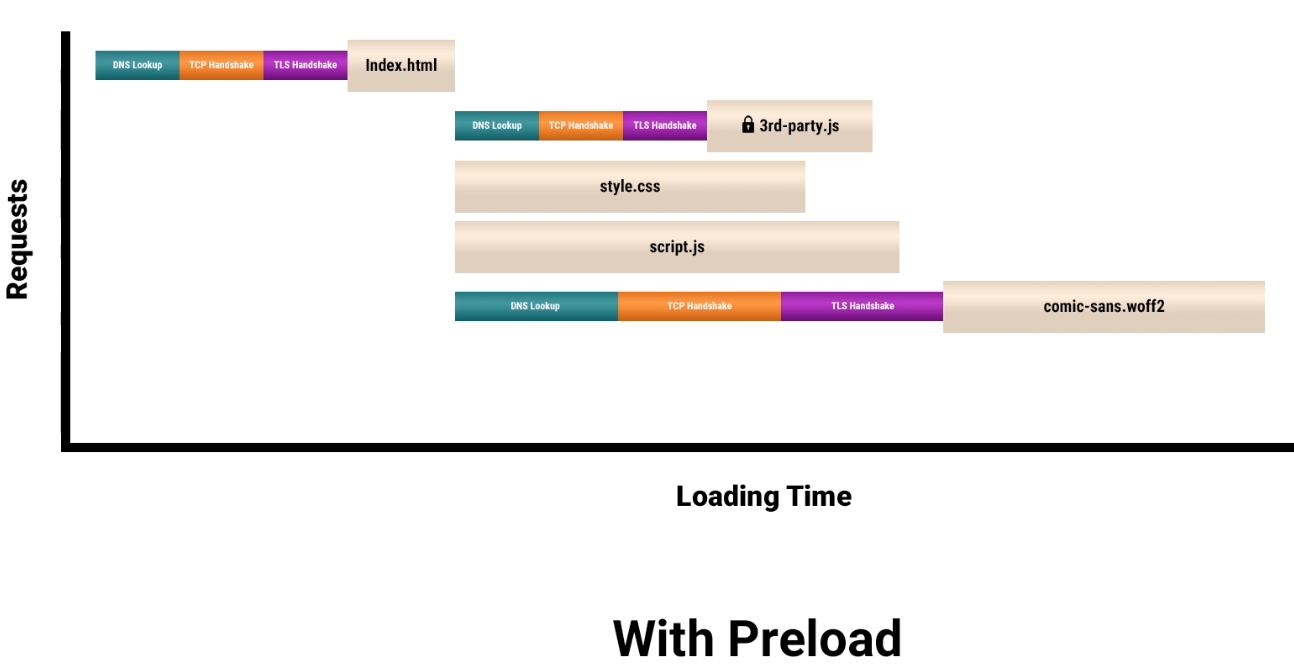


Also it’s worth noting, that as of the time of recording Safari doesn’t yet support Prefetch resource hints so any benefits on that browserwill be lost.

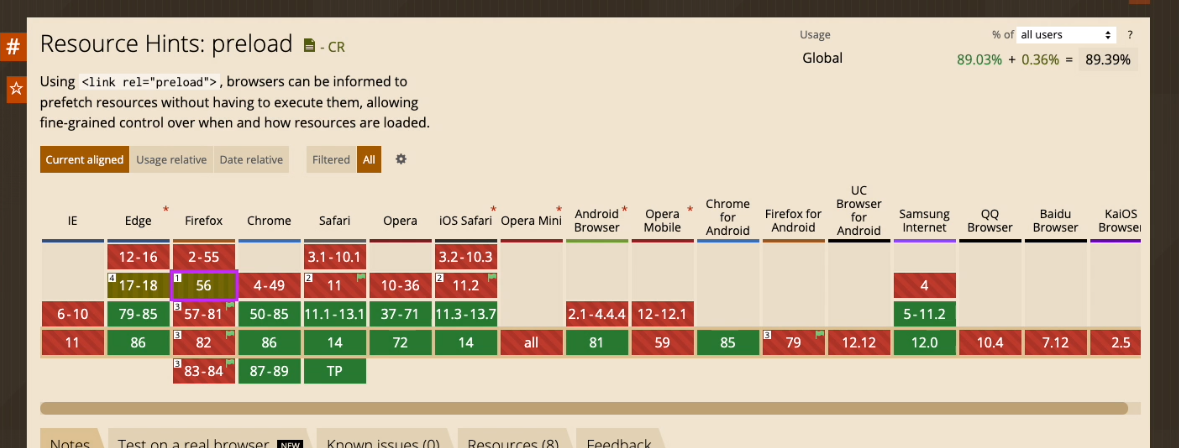


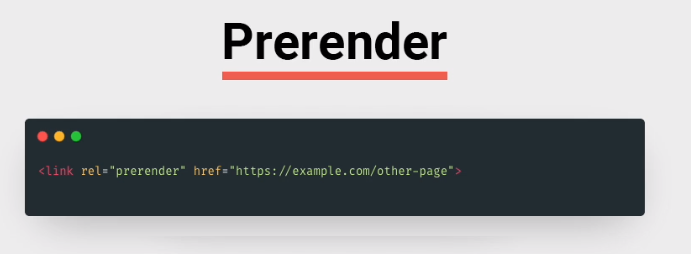
Preload is very similar to Prefetch, except that it’s no longer a suggestion, it’s an order.

The browser must download the file referenced in the resource hint regardless of what it thinks, straight away.



Again, this isn’t supported in all major browsers. At the time of recording, Firefox doesn’t support this resource hint.





Prerender is the most extensive resource hint and can only be used with other web pages.

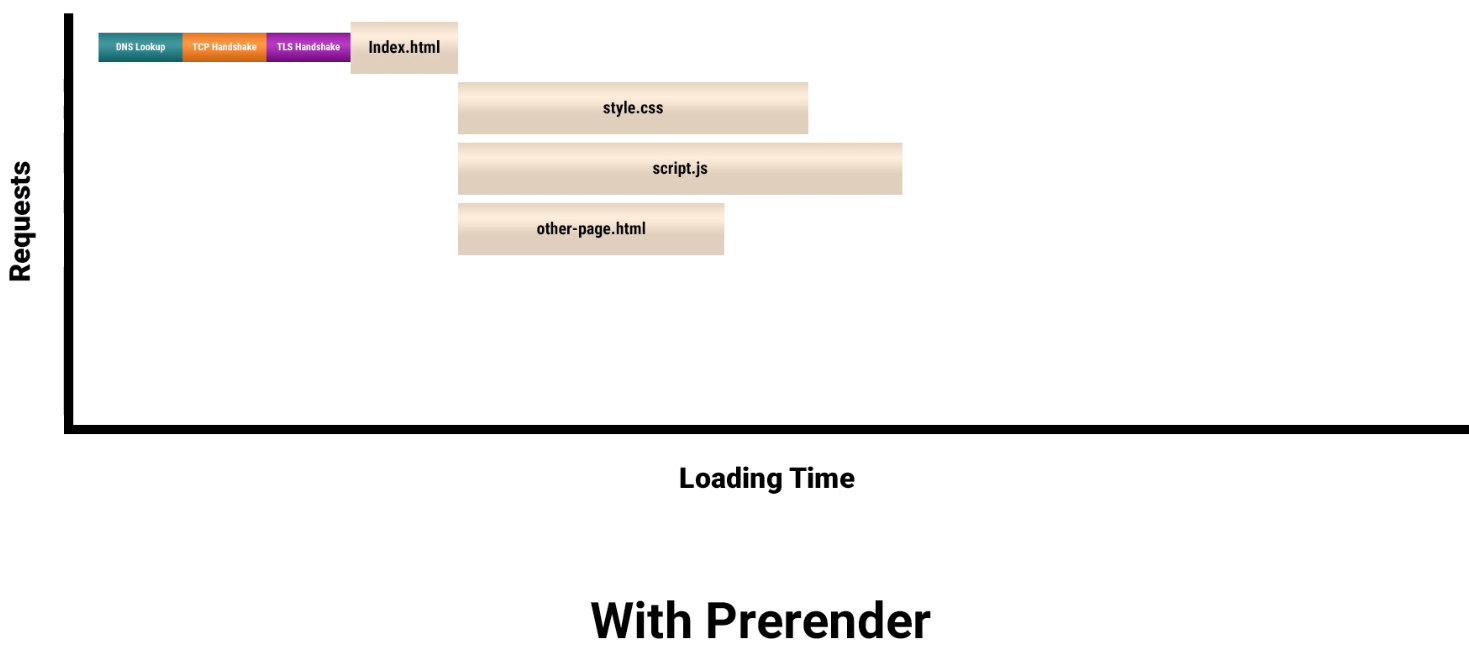
In addition to opening a connection and downloading the page’s HTML, the browser goes on to parse its contents and then download and execute it’s discovedred resources as well.

Essentially, the page is rendered in a hidden tab in anticipation of the user visiting it.

Unfortunately if this doesn’t happen, then all the efforts on the current page will have gone to waste.

This means Prerender is best used where you can predict the user’s next move with a high level of accuracy, like a results page following a search or multistep checkout journey going in a single direction.

Currently, as far as major browser support goes, Prerender is currently only supported on Chrome and strangely, Internet Explorer 11.



Example,

Insert the following code snippet into header.html

{% if path != ‘/product s’ %}

<link rel=”prefetch” href=”/products” as=”document”>

{% end %}

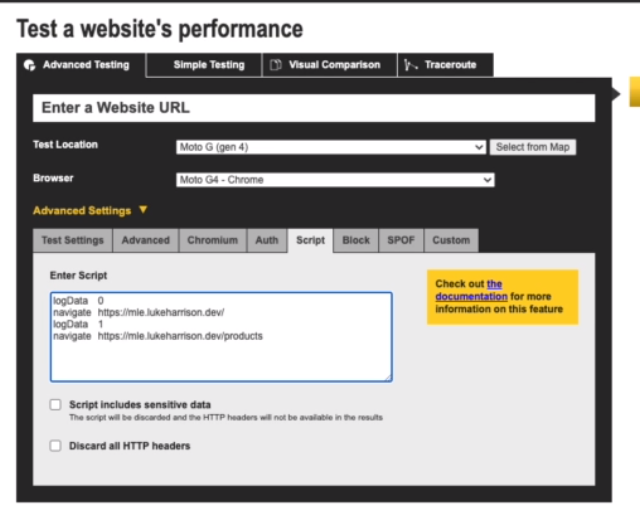
First off, it's important to measure the impact loading this extra resource has on the home page, as if it’s slowing it down for relatively little gain on the products page, the trade off won’t be worth.

Here’s a comparison of the home page with and without the prefetch.

As you can see, the difference is negligible. And as for the product page itself, the prefetch has meant that it renders nearly half a second early. Looking at the waterfall, you can see why. As the browser already cached the HTML back on the homepage, we no longer need to download it on page load and can focus exclusively on uncached assets like the images.

To capture this data meant using WebPageTest’s scripting capabilities.

Here on test config, we’re telling WebPageTest to first load the homepage, which triggers the Prefetch before then navigating to the products page.



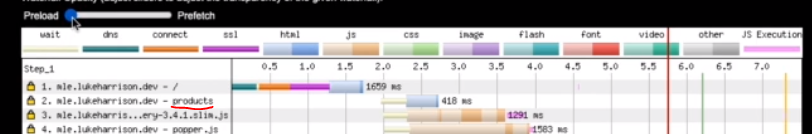
We use the logData function to suppress data logging here until we’re on the products page.

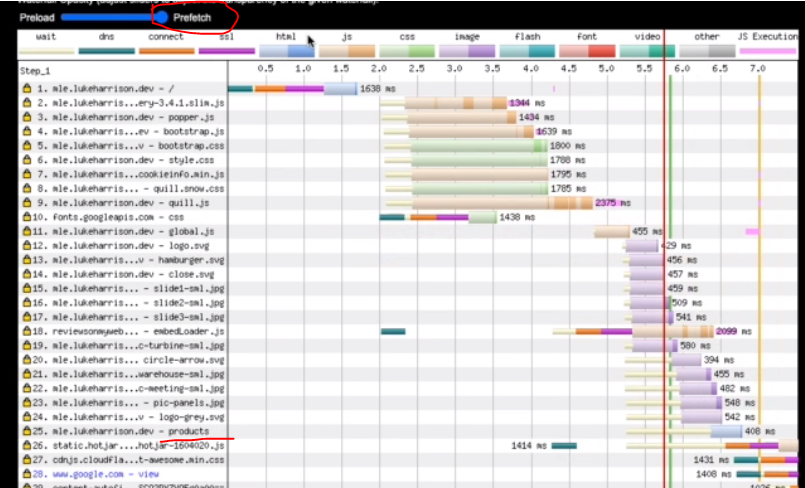
Before we wrap up, you may be thinking, why not use preload in this scenario? This would guarantee that the product page will always be downloaded early, right? This is true.

And when comparing Prefetch and Preload’s impact on the products page, the results are pretty identical. The difference though, is back on the homepage.



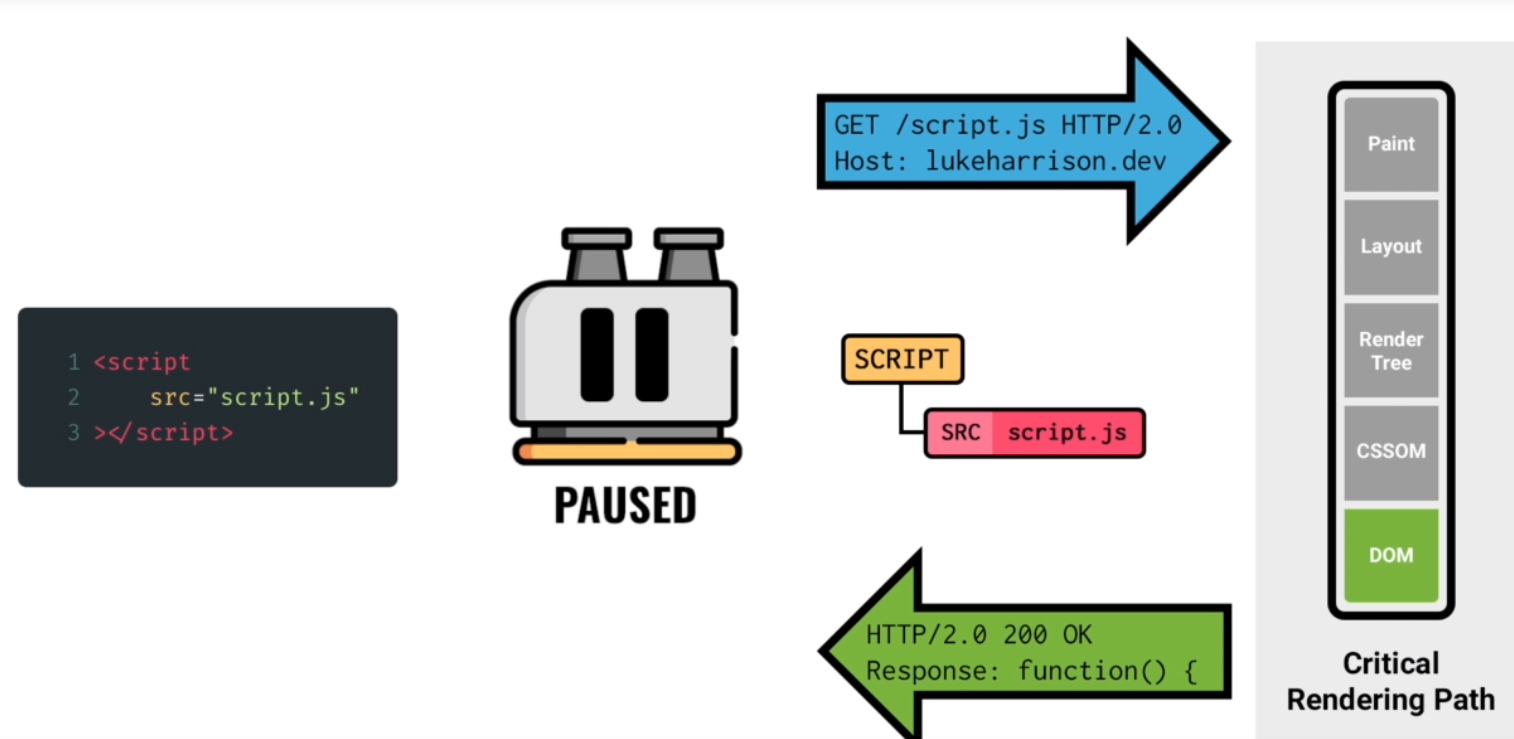
The homepage with Preload renders half a second later than with Prefetch. This is because the browser treats Preload as a command rather than a suggestion and is prioritizing it over loading the page’s own critical resources.

  
Contrast this to Prefech, where the products page is downloaded with a much lower priority.



8-30 Async / Defer Javascript

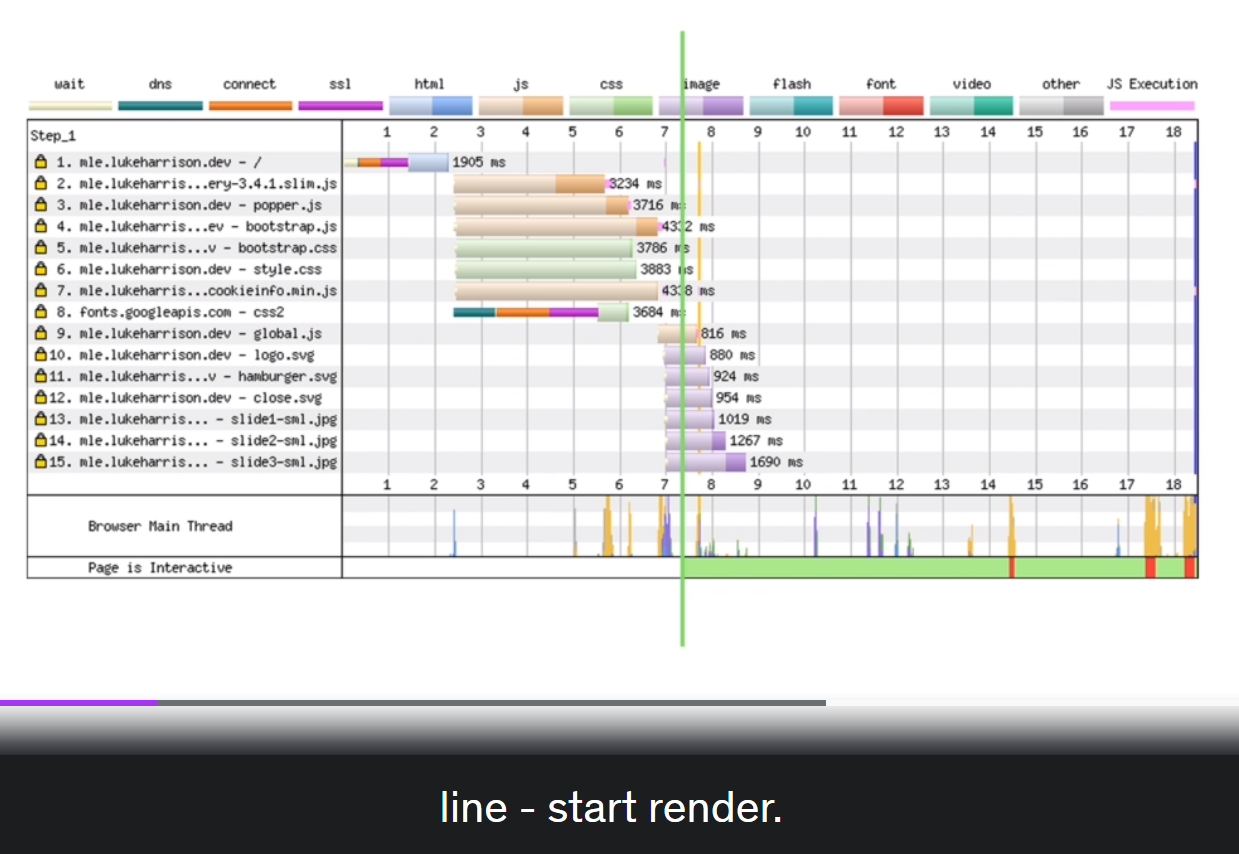
If we think back to the "How Browsers Load Websites" video, we learned that as part of the critical rendering path, the HTML parser builds the DOM, if during this process it finds a script element linking to a JavaScript file, parsing is paused until this request is completed and the script executed.



This makes JavaScript a blocking resource as it delays rendering.

We can see this on the website's waterfall here.

All the JavaScript, these yellow bars, are downloaded and executed before we reach this green horizontal line - start render.



Thankfully, modern browsers provide us with 2 attributes to tackle this issue.

The first, defer, delays JavaScript execution until after the HTML has been fully parsed.

This means the HTML parser pause is no longer required, meaning the script is no longer render blocking.

The second, async, is very similar, except that JavaScript execution happens as soon as it's able to,parallel to the HTML parsing.

Async is best suited for JavaScript which doesn't require a complete DOM or JavaScript without a dependency on other scripts having loaded.

Let's remove the render blocking JavaScript from MLE power.

Open the head template partial, these scripts all manipulate the DOM in some way or are dependencies of cripts which do, so we'll set these to defer.

This one, which injects a cookie banner, can be set to async. It only adds to the DOM, rather than interacting with existing elements, so doesn't require it to be complete.

Whilst this one, within the layout partial, contains miscellaneous DOM interactions which are scattered across all pages.

Prior to async and defer, its placement here would be valid.

However, now we can control when JavaScript is executed, we can safely append this to the head and set it too to defer.

I'm mainly expecting to see improvements to the initial rendering, which looks to have happened.

With JavaScript no longer blocking the critical rendering path, it's able to complete much sooner.

Nearly 20% sooner for Start Render, First Contentful Paint and Largest Contentful Paint.

And as we know, where rendering metrics go, Time to Interactive follows. We're seeing a 34% increase here, which is definitely welcome.

Finally, a nice boost of 17 has been added to our Lighthouse score.

# 8-31 Text Compression

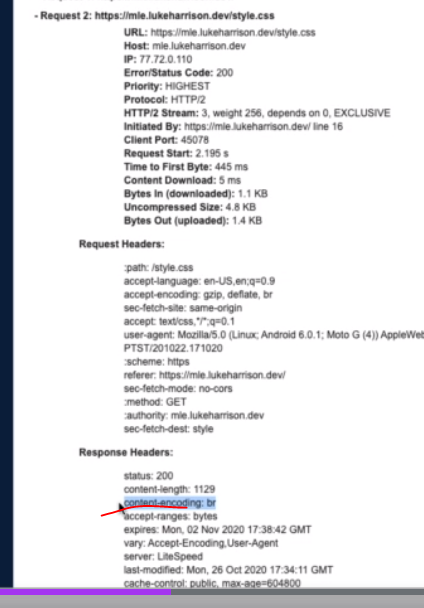
When text assets like HTML, CSS and JavaScript are sent from the server to the browser as the website is being loaded, you can compress these files using special file formats.

The oldest and most widely used of these is **GZIP**, which has been around since the earliest days of the web.

The other widely used format is **Brotli**, which was developed by Google and allows you to achieve even greater levels of compression.

Typically, the larger the file, the higher the compression level.

You can see if a resource is being compressed in devtools.

WebPageTest also shows you here.

If you on shared hosting like MLE Power, the chances are good these will already be enabled.

If you manage your own hosting infrastructure, then both GZIP and Brotli are supported by server software like Apache and nginX.

How much of a difference does compassion make to the loading of a website? We can find out using WebPageTest where we'll run 2 performance audits on our website, one with compassion enabled and one without.

I can disable compression by setting this header on all HTTP requests.

So here's the comparison, the uncompressed variant takes around a second longer to start render, and much, much longer to fully load.

You can see by comparing the waterfalls, that things are just generally taking longer to complete.

This is reflected in these timings here where many of the key metrics are as much as 50% faster when text assets are being compressed.

And this is why. Going from compressed with GZIP and Brotli to uncompressed means 275% more bytes are being transferred on page load.

This is best seen in video form, where the large gaps between the 2 are pretty evident.

With JavaScript accounting for the bulk of the bytes here, imagine if it were still blocking.

# 8-32 Text Asset Optimisation

Currently, the website's CSS and JavaScript is split across multiple files.

By bundling these into single files, we'll help the browser to download them faster.

This is because compression formats like gzip and Brotli work better on larger files, so even though we're downloading the same code either way, our bundled files will still be a fewer in bytes.

At the moment, our third party files are sitting in the root www directory.

Where possible, we'll swap these for NPM packages, which in the long term is better for things like version control.

We'll make sure to install the same package versions as we are currently using to make sure the code is exactly the same.

To bundle our CSS into a single file, we'll be using SASS, the CSS preprocessor, which extends the language with more programmatic capabilities.

Create a new directory in src called scss, and style.css from the root directory to here.

Rename it to style.scss, which is the typical SASS file extension.

Inside the file, we can import the CSS components for Bootstrap like this.

Because SASS is a preprocessor language, it needs to be compiled into CSS.

Thankfully, this kind of thing is what gulp was made for.

In the command line, we need to install the SASS dependencies,

then in the gulpfile, let's import them so they're available to our gulp tasks.

And now let's create the task itself.

This takes the sass, which now sits in the src directory, compiles it into CSS and then drops it in the root WWW directory, where the website will expect it to be.

The compressed setting here means SASS will remove all line breaks and white space from the CSS, which has an impact on the file size of the compiled file.

Now, we just need to add this to the build task and we're ready to go. gulp build.

And here's the file. Looks fine.

Let's delete these CSS files in the root directory as they are now being imported instead. We need to remove the link elements from the HTML as well.

One final thing. In the gulpfile, update the clean task to remove any CSS files from the root directory before a build to make sure any old files are purged.

And then in the command line, rerun gulp build.

Now if I view the website, everything looks the same, which is good news.

Let's look at the amount of bytes we've saved. Before, the total weight of the then separate CSS files came to 195kb, which came down to 36kb if compressed with gzip or 24kb using Brotli.

Now we have our bundled CSS, which weighs in at 165kb.

If gzipped, this comes down to 32kb or with Brotli, it comes down to 22kb,

so a saving of just a few kilobytes. It may not seem like much, but you have to think that as the website grows and develops, this gap will only widen.

Create a new js directory in src and move global.js also cookieinfo.min.js from the root to here.

The latter of which doesn't have an NPM package, so we're forced to use the same file. As for every other JavaScript file here, delete it.

In the global.js file, import the other scripts at the top, including the cookieinfo script.

Moving on, like with most things in web development, there's plenty of ways to bundle JavaScript. Here, we're going to use rollup.

To get started, install these dependencies.

Now open the gulpfile and import the dependencies at the top of the document along with the rest.

And as for the gulp task itself, here's the code.

So here, we're using rollup, the JavaScript bundler, to grab the global.js file from source.

Next, the contents of the new file are passed to a plugin called nodeResolve, which lets us import packages from node\_modules.

Then they go to terser, which is a function that in addition to removing line breaks and whitespace renames functions and variables where possible to something shorter without changing the behavior of the code.

This can have a huge impact on the final size of the bundle as JavaScript does have a tendency to just grow and grow in file size as websites grow and develop.

Finally, the bundled JavaScript is sent to the root www directory.

As before, we just need to update the clean and build tasks.

In the HTML, we can now also remove the script elements which reference the now bundled JavaScript.

Finally, if we run gulp build, we'll see some additional log output and then in the root directory we'll see the bundled minified JavaScript file.

Again, let's run some basic comparisons to see how many bytes we've saved.

So before, when each script was separate, together they came to 686kb uncompressed, 190kb with gzip or 151kb with Brotli.

Bundled, it's 158kb uncompressed, 57kb with gzip or 48kb with Brotli.

So, at least 50% smaller then. A big saving.

We've changed a lot this time around.

Instead of loading multiple smaller files, we've combined our CSS and our JavaScript into single bundles.

Because compression formats work better on larger files, our bundled code is now able to download quicker than if it were separate, meaning the browser is able to process them much sooner.

The same applies for Time to Interactive, which has now more than halved, beating our goal for the metric.

If you have a fast rendering website and a faster loading JavaScript file, that's the perfect recipe for improvements here.

Overall, the Lighthouse score has increased to 81, so we're not too far away now from the green.

# 8-33 Critical CSS

If we look at MLE Power in waterfall view, CSS is now the only remaining render blocking resource preventing the critical rendering path from completing immediately.

This is CSS's default behavior, as if page render wasn't blocking until CSS is downloaded and parsed, the page would render before the browser knows how to style it and you would experience what's called flash of unstyled content.

However, there's a way to stop CSS blocking page render and sidestep the flash of unstyled content issue.

They 'have your cake and eat it' approach is done by determining the bare minimum CSS required to style the content in the initial viewport, as that's all the user sees as the page is loading.

We then separate this critical CSS from the rest and display it inline in the HTML document directly.

The remaining non-critical CSS stays where it is, in the CSS file, which is still cachable by the browser.

Now the browser has everything it needs to render at least the viewport content correctly, without having to download any additional CSS files.

This will solve the flash of unstyled content problem, at least from the user's perspective.

The remaining non-critical CSS is then loaded asynchronously at a later stage, which, like with async or defer on script elements, removes CSS from the critical rendering path entirely, solving the render blocking problem.

So, how do we separate critical CSS from noncritical and make the latter loaded asynchronously?

Thankfully, there's an NPM package called Critical which can automate this for us.

Remember to then import it into the gulpfile.

And now he's the task.

First off, we used a glob function to grab all of our compiled HTML files from the root directory as one page's critical CSS is different to the next.

This gives us an array of paths to loop through, however, because the critical module doesn't expect paths, only filenames, we have to use a native node module called Path to separate these.

This is imported the same as any other module, but because it's native to node, we don't need to install it as a dependency first.

We then run the critical function and pass it some data.

Inline is set to true, which tells the function we'd like to inline the critical CSS in the HTML for us. False on the other hand, tells the function we'd just like the critical CSS by itself.

We set base to the root www directory.

Source refers to the HTML which we'd like to generate critical CSS for, in this case the current page in the loop.

And target is where the output of all of this should be sent to.

Again, the HTML.

Next, there's the width and height properties.

These correspond to the dimensions of the browser window for when critical CSS is captured.

Here, it's set to desktop level sizes, but because the CSS is written mobile first, critical CSS for mobile will also be captured as well.

So all device levels are covered.

Finally, there's Ignore, which gives the function a list of CSS which it should not extract.

Included here is font-face, because we're currently using Google Fonts, which has its own third party CSS.

If the function were to extract CSS from here, because it's hosted externally, it wouldn't be able to remove the critical CSS from the original file, so you'd have duplicate rulesets.

Just need to add it to the build task and away we go.

Now if we go to the compiled index.html, here's the inline critical CSS, now separate from the rest of the CSS, which is still loaded here as before.

Well, not quite as before, as some additional properties have been added, which we'll get to shortly.

But first, if we open this in the browser, everything looks fine, which is good.

Next, back in the HTML, delete this link element temporarily. When I reload the page, again, everything still looks fine.

However, when I scroll, suddenly everything goes wrong. This is because the only CSS loaded now is the critical CSS and the critical CSS only contains styles which are needed for the initial viewport.

The remaining CSS is noncritical, which we're now loading in a non-blocking manner, thanks to these attributes which have been added by the critical function. Essentially, this is a bit of a hack.

We're telling the browser here that the CSS should only be applied when the webpage is printed, like out of a printer... to paper.

Therefore, it doesn't consider the CSS important enough to wait around for, whilst it's downloading, as it styles wouldn't be visible on screen anyway.

Then, when the stylesheet is downloaded, having not blocked rendering, it's onload attribute is used to switch its media attribute to 'all'. The browser spots this, so then applies the CSS styles to the elements on the page.

It's an ingenious little technique,

so let's see how much of an impact it makes to performance.

Basically, it makes a huge difference. Start render is nearly a second and a half faster, you can see why in the waterfall.

Comparing the 2 views, HTML is now the only remaining render blocking resource, with everything else loading asynchronously as the HTML parser discovers it.

It's not all good news though, it still takes 4 seconds to display the logo, the hamburger icon, and then the first carousel slide, so for most of the loading time we gained, the user is sat staring at what amounts to the website's scaffolding.

Let's use the Preload resource hint to force the browser to load these critical images sooner.

That way, we should be able to improve our website's perceived performance, even if it doesn't impact the actual performance at all.

We'll preload the logo and the mobile hamburger icon in the head partial, as these are shared across all pages.

As for the carousel slide, this is only needed on the home page, so in the page template, we need to add a Nunjucks head block containing the preload, like so.

Because there's 2 variants of the slide, a smaller one for mobile and a regular sized one for desktop, we can add a media attribute so the browser knows when to preload each.

Now, with the preloads in place, let's do another comparison.

Much better, the logo loads first followed quickly by the hamburger icon and the carousel slide.

So in terms of perceived performance, instead of feeling complete at 4.1 seconds, this has now been pushed forward to 2.9 seconds, which is a significant jump.

Even though the visually complete metrics are a little slower, when you see the comparison video, it doesn't feel like it, does it?

Think back to the Amazon video from the earlier perceived performance video.

This is the same kind of thing, albeit on a much smaller scale.

By removing CSS from the critical rendering path and giving the browser just enough inline CSS to render the initial viewport, we've made huge improvements to our rendering times and flown past our 3 second goal.

Given this, I would have thought we'd see an improvement Time to Interactive as well. Perhaps we've reduced this as much as we're able to now through improvements to rendering speed alone.

npm install –save-dev critical

<link rel=”stylesheet” href=”style.css” media=”print” onload=”this.media=’all’”

# 8-34 Google Fonts Optimisation

Google Fonts is the most popular way of including non-standard fonts on a website. Out of the box, in 2020, its performance is already quite good.

You add them to a website through a script like this.

Looking at our website's waterfall, we can see the request here, which has already been made non-blocking automatically by the critical module from the previous video.

The request returned CSS font-face rules, which the browser can then use to download WOFF2 files, which on modern browsers, is the format most commonly used for web fonts.

These are then applied to the pages text, provided the correct font-family CSS is in place.

Still, there's a bit of room for improvement.

The Google Font files are what's called the late discovered resources.

This is because currently the browser has to wait until the Google Font'ss CSS is downloaded and parsed before it discovers that the font files even exist.

Whilst we can't preload the WOFF2 files directly, as the file names will change often as the fonts are updated by Google, we can at least Preconnect to the domain, which won't change, to get the network negotiation

out of the way as early as possible.

The WOFF2 fonts come from a domain called fonts.gstatic.com.

Lets Preconnect to this.

Now, looking at the waterfall, with the change in place, it doesn't seem to be doing anything.

This, it turns out, is because when Preconnecting to font resources, you need a crossorigin attribute on the link element to satisfy the font face specification.

Now if we look at the waterfall again, the Preconnect looks to be working and compared to before the fonts starts downloading immediately once the Google Fonts CSS is downloaded, meaning those web fonts will appear on the page much sooner.

Another popular recommendation to speed up Google Fonts is to preload the initial CSS like so.

In this case however, if we compare the waterfall with and without Preload, there's little difference.

This is because the only blocking resource here is HTML,

so any subsequent requests are fired as soon as the HTML parser discovers them.

So essentially, there's nothing to Preload as the browser in this instance is already downloading this resource immediately.

It's worth remembering, though, that MLE power is still only a relatively simple test website.

On much bigger and more complicated websites, preloading Google Fonts will likely make a difference.

# 8-35 Self-Hosted Fonts

Currently on MLE Power, we're using Google Fonts.

There's an alternative to this, hosting them ourselves. Self hosted fonts open the door to more optimizations that just aren't possible with third party hosted fonts, as we now have more control over how these fonts are downloaded during the loading process.

Let's make the switch.

First off, create a new directory in the root called fonts. Our self-hosted fonts will live here.

Next, we need to download Lato, the Google font we've been using.

How do we do this? One way is the same as the browser, grab the links from the Google Fonts CSS and download each file one after the other.

This can be a little cumbersome though, so my preferred way is to use this helper app.

Search for Lato, grab the fonts we need, which in this case is regular, regular italic, bold and bold italic.

Make sure only Latin is ticked here.

This means only the basic font characters will be included, reducing the filesize of the font file.

Removing characters from a font you don't need is called subsetting.

Here's the font-face CSS which we'll need to copy and paste into our own.

By default, this includes a very browser compatible list of font formats, many of which in 2020 just aren't required.

By clicking this modern browser tab, it reduces the file formats down to just WOFF and WOFF2, which are supported by modern browsers all the way back to Internet Explorer 9.

Change this path here to match our own directory structure, then copy all of this and place it in your style.scss file towards the top, like so.

Finally, back on the web page, download the fonts using this button and put them in the fonts directory.

For your convenience, you can also just download the CSS and the fonts from this video's attachment in case this neat tool ever goes offline.

Now, let's go back to the CSS we've added.

font-face rules allow us to define our custom fonts.

In this first instance, we're declaring a font called Lato, which will be shown when the CSS font style is set to normal, with font weight of 400, which is typically the browser default anyway.

We're then linking to the font we would like to show when these conditions are met, which are downloaded via these urls here.

You'll notice however, that there's a local function here referencing our font name.

All this does is tell the browser that if the user already has this font installed locally on their machine, then don't waste time redownloading it.

Now we've got our local fonts in place, let's remove all references to Google Fonts from our head template.

Follow this up with a gulp build and voila.

Refresh the page, and if you inspect the code, the Lato font should now be self-hosted and not coming from Google Fonts.

So here's the waterfall for Google Fonts versus self-hosted fonts.

A concern here is that with self-hosted, the fonts are being loaded much later in the process, which is leading to a worst visually complete metric, which leads to a worse perceived performance, despite reducing the overall load time.

If we look at this in video form, Google Fonts loads in at 4.5 seconds versus 6.7 seconds for our self-hosted ones.

Why is this happening?

Well, for one, we're not including our font-face CSS in our inline critical CSS.

This is because we added an ignore rule in the gulp task to prevent duplicate rulesets.

This isn't a problem with self-hosted fonts, as we're in control of everything, so let's remove the ignore rule from the critical gulp task.

Does this make a difference?

Yes, it does.

The fonts are now discovered and thus loaded much earlier.

We've also improved upon the visually complete metric from before, beating even Google Fonts now. This looks good.

Let's check the timeline.

Well, the good news is that the self-hosted fonts now appear at 3.5 seconds instead of 6.7 seconds, which is a huge improvement.

The bad news is that up until this point, there's no text.

This problem is called flash of invisible text.

It's happening because of how browsers load custom fonts, which can be controlled by a CSS property which works alongside font-face called font display.

By default, most browsers briefly hide text until the web font has been downloaded.

This is the equivalent of setting font display to block.

The value we want here is swap, which instructs the browser to display text using the next available fallback font until the primary choice, the web font, is fully downloaded.

The fonts are then swapped. The period where the fallback font is used is typically called the flash of unstyled text.

Let's add it to our CSS font-face rules.

You may be thinking, why didn't this problem occur when we used Google Fonts?

That's because if we look back at the Google Fonts CSS, the property is already there.

Now here's the comparison, we have the original Google Fonts version, the self-hosted fonts with the invisible text problem and now the font display version.

Thankfully, that seems to have fixed the issue.

If we look at the video, the text swap still happens at 3.5 second point, but instead of swapping invisible text for our web font, we're swapping from the fallback font.

Now that our font URLs are static and won't change, we can also preload them.

This will make the browser download them as soon as it's able to, which should make the swap from fallback text to web font happen sooner.

Add the preload resource hint to the head template like so.

The type attributes here is used to tell the browser that if it doesn't support this format - WOFF2 - it can ignore this preload.

In that instance, it would then hit the second preload, the standard WOFF format, which it will support.

Let's see if the preloads have any effect.

With preload, the font swap from fallback to main now happens at 3.3 seconds point instead of the 3.8 seconds point.

That's good.

What's bad however, is that elsewhere in the comparison, every other major metric is actually worse, the most important of these being the page's perceived performance. Because we're now preloading fonts,

the browser clearly considers these to be more important, so downloads them first.

This means even though both variants have comparable loading times, the one where we're preloading web fonts feels like it loads much, much slower. 2 seconds slower to be precise.

Personally, I don't feel like the trade off is worth it, so let's remove these web font preloads and revert to before.

In some instances, preloading web fonts may yield more positive results, likely on much bigger websites with much more assets to be downloaded.

The important thing is this - test or you'll never know.

System fonts are the funds already in use on your device's operating system. Using these on your website can grant a little performance boost because you don't have to download any additional web fonts,

and also, as a bonus, they can look pretty cool.

In 2020, lots of websites are using them as their default font stack. GitHub, WordPress, Medium and Ghost to name a few. To use them, you typically add a font-family CSS rule like this.

There's lots of fonts listed because no one operating system has them all by default. If an operating system doesn't have one font, it simply moves on to the next and the next until there's a match.

Bootstrap the framework also uses system fonts by default.

This means to enable them for MLE Power, which imports Bootstrap,

we just have to delete these style blocks from our CSS.

Might as well also delete the web fonts from the root directory as well.

Here's how the website looks when viewed on a Mac. For comparison, here's same view with the Lato web font

we've been using. Not a massive difference and certainly not bad by any measure.

How does this compare from a performance perspective?

Because we're already loading our self-hosted fonts quite efficiently,

the only major difference is with the perceived performance, as there's no longer a sudden change of font halfway through the loading, which is certainly a positive.

The overall loading time is also slightly less as we're no longer having to download extra web fonts.

So the question is, which do you go with, web fonts or system fonts?

The answer is, as always, it depends.

If you're in control of the design and you don't mind how the system fonts look across major operating systems, then great, go for it, though, if you're working on someone else's brand, then it's likely that you'll need to use specific fonts to adhere to it.

In this case, self-hosted fonts are the only solution.

For MLE Power, we're going to stay with our self-hosted Lato font, as when the practical section is complete and we compare the initial website to the finished article, it'll be a more accurate comparison.

The biggest gain here is with a metric that we aren't actually tracking, the point where the fallback font swaps over to the web font. With this new set of performance tests, before, with Google Fonts, the switchover happened at the 5.3 second mark, whilst now with self-hosted fonts it happens at 3.7 seconds.

Elsewhere, we're also seeing a hit to some rendering metrics. Looking at the timeline though, this seems more because the server was running a little sluggish for that run, as everything seems to be taking a little longer to download.

Meanwhile, there's a slight improvement to Time to Interactive. This is too small to be attributed to anything specific, though.

# 8-36 Lazy Loading

By default, when this image heavy products page is downloaded, so too are all of these images, whether the user sees them or not.

This is definitely a big waste of bytes. The solution is lazy loading.

Lazy loading allows us to defer the loading of assets until they're in the viewport and visible to the user.

This means we only download what we need, like in this example, where we can see each image request firing as I scroll down the page and discover new images.

Making an image lazy load in Chrome, Edge and Firefox is as simple as adding the loading attribute and setting it to lazy like this.

Now, if I reload the products page following a build, I can see in the network tab that these product images are now only being downloaded as they approach the viewport, which is good.

What about other browsers?

This is where it starts to get a little complicated.

Native lazy loading at the time of recording is only supported by Chromium browsers and Firefox.

This means for Safari and older browsers, you can either not implement lazy loading and have it just fail gracefully to a regular load or implement lazy loading via a polyfill.

If you select the second option, it's currently not possible to polyfill this functionality 100%.

The closest we can get is by using this polyfill, which uses the browser Intersection Observer API to reproduce the functionality,

which itself needs a polyfill if you're needing to support Internet Explorer.

Install the lazy load polyfill as an NPM module like so, then import into our global JavaScript file.

Back on the products page, to get the polyfill to work on browsers like Safari, the images need to be wrapped in a noscript element like so.

Now, if I rebuild and then reload the page in Safari, there we go - lazy loading.

iFrames can also be lazy loaded. On the homepage way at the bottom, there's a Google Maps embed which uses an iFrame.

Normally, the scripts which power this would always be downloaded and executed, even if the user never sees it.

Using the same attributes as with images, followed by a rebuild and a reload, these scripts are now only loaded as the iFrame itself is about to appear within the viewport. Like with images, this has the potential to really minimize the amount of bytes downloaded on first load.

It's worth noting that only Chromium browser's currently support this, as Firefox only supports lazy loading for images.

Thankfully, the polyfill we're including also takes care of this for us.

One thing noticeable, especially on slow networks, is the amount of judder as images are loaded in, lazy or otherwise.

The rate of shifting elements can be measured using the Cumulative Layout Shift metric, which was first mentioned way back in the loading performance metrics video.

To stabilize this, we can add width and height attributes to our website image elements, like so.

In modern browsers, in addition to setting size, these attributes are now used to calculate an aspect ratio so the image element can be appropriately sized ahead of the image loading.

These values may not represent the exact size of an image, but ultimately it doesn't matter anyway.

200 pixels by 100 pixels on mobile or 500 pixels by 250 pixels on desktop, the aspect ratio is the same.

Though a minor bug - notice how the image size is now warping as the browser scales down.

This is due to the absence of a generic CSS height rule on image elements, making the browser use the height attribute instead.

Add width and height attributes to the rest of the website's images.

In Chrome, you can easily discover the pixel values like so.

This layout issue also applies to video elements as well, so make sure to give them width and height attributes.

Let's finish off by doing a comparison before and after lazy load and size attributes.

We'll test the products page, which is image heavy, so stands to gain the most from these changes.

Right away, in the waterfall, look how fewer image requests there are when lazy load is active.

The overall low time is reduced, as is the number of requests and bytes loaded, that's to be expected.

There isn't much in the way of Cumulative Layout Shifting, this could be because the layout is quite simple.

If we run the same kind of comparison on the homepage, which has elements positioned in more complex layouts, you see a different story.

Far fewer layout shifts were recorded when images had width and height attributes, which is why it's dropped to almost zero.

We've also cut a significant chunk from the bytes downloaded metric here aswell, as we're not immediately downloading all of our images now or the contents of the Google Maps iframe.

So then, it's clear that lazy loading and setting width and height attributes has a positive effect on aspects of the performance. Apart from the overall Lighthouse score, which now sits at 82, we've now also hit all of our performance goals.

npm install –save-dev loading-attribute-polyfill

# 8-37 Remove Unused CSS

At the moment on each page, the browser is downloading and parsing quite a large CSS file.

In Chrome devtools, there's a tool called coverage which can show us how much of this is unused.

Here, we can see that in style.css, 93.3% of CSS styles aren't being used, which is crazy! Let's remove unused CSS from our stylesheet, so each page only downloads what it actually needs.

Each page is going to need its own CSS file, as the CSS one page uses is going to be different to the next.

We'll name each CSS file according to the page it's used for, so for index.html, there would be an index.css for example.

Begin with installing and then importing this dependency - gulp-data.

We're going to use this in our Nunjucks task to pass the template its own file name.

Then in the head template, replace the reference to style.css with this filename variable.

This means, on compile, the page will look for a CSS file matching its own name.

Next, we need to create the CSS files. At the moment, our compileSASS function is only creating one - style.css.

We need to duplicate this for each page.

This can be done using the glob function, which we're using here to grab all of our HTML files.

We're then looping through them and for each one compiling a CSS file and then naming it after the current page in the loop.

Now, on build, it's generating multiple CSS files, currently all the same.

All that remains is that for each page's CSS file, we remove any CSS which isn't being used.

We can automate this process using a module called PurgeCSS.

Start by installing and importing like usual.

And then create a new gulp task called removeUnusedCSS.

Like before, we need to grab our HTML files using glob and loop through them.

Here, we'll use the PurgeCSSmodule like so.

For each CSS file in the root directory, we're passing it to PurgeCSS and then pointing it towards its corresponding HTML file, which has the same file name.

The module then uses this link to determine which CSS is being used and updates the CSS file accordingly.

Let's add this to the build task like so and then run.

Right, this doesn't look good.

What is happening here is that the removeUnusedCSS task is trying to run whilst the criticalCSS task is still in progress. At the moment, I haven't found a genuine fix for this, as I am by no means a gulp expert.

I have found a workaround though.

Remove the unused CSS task from gulp build and then import it into its own separate task here.

Next, open up package.json.

We'll create an NPM script here to alias our build task.

Think of these as shortcuts which you can create to run other commands.

We can check it's linking OK by running 'npm run build' in the command line.

Now, create a new and NPM script called postbuild.

By naming this after the build task, with post as a prefix, it will automatically fire afterwards.

In here, we'll run the unused CSS task.

Now, run the build task again.

Brilliant.

Yes, it's a bit of a hack, but it's the only way to get the modules to behave themselves and run sequentially.

Looking at the CSS in our root directory, it's being cut down in size significantly.

Let's check the web site.

That's not good.

The carousel is broken and these drop downs are missing their transitions.

What's happening here is that because PurgeCSS only looks at the raw HTML, rather than the rendered web page, dynamic classes normally inserted into the DOM by JavaScript are being incorrectly flagged as unused and thus removed.

We need to revisit the removeUnusedCSS gulp task and get to the module to ignore these dynamic classes.

This can be done through PurgeCSS's safelist property, which accepts an array of regex patterns.

As each page is different, we'll need a separate safe list for each page. Create an object called safelist up here and then pass it to the safelist property like so.

This means as the purgeCSS cycles through each of our HTML files, it will look for a safe list matching the filename.

Only the home page has dynamic elements on it, so only 1 safelist is needed, like this.

Now, when we run PurgeCSS, CSS classes on the home page matching these patterns will be ignored.

So when we revisit the page, it's all then looks normal again because the wrong CSS classes haven't been removed.

Before this home page was at 93.3% unused.

Now it's at 44%, which is a big improvement.

Remember, this is never going to be 0 as some styles are only ever used at mobile resolutions for example, which is fine.

How does this affect the performance?

There's no real change to the rendering performance as the CSS we're filtering is no longer blocking.

A millisecond here or there isn't anything to worry about.

The biggest change is here though, Bytes Downloaded and then Fully Loaded, as the amount of CSS downloaded has been reduced by quite a bit.

Finally on Google Lighthouse, we've achieved a performance score of above 90.

Hitting the green was one of our key goals in this practical section, so it's nice to finally achieve it.

npm install –save-dev gulp-data

npm install –save-dev gulp-purgecss

# 8-38 Remove Unused JavaScript

Now that we've removed unused CSS, we can move on to JavaScript. To keep the focus on web performance, we'll try to keep this as simple as possible and use a similar approach to the one taken with CSS, albeit a little more manual.

Instead of having one global JS file, which every page downloads, each page will have its own file containing only what it needs.

Let's start by creating an index.js for our homepage.

Next, we need to split the existing scripts.js file into smaller partials organized by feature, so there can be easily imported where required.

Create a new directory in the src/js directory called partials.

Now back in the JavaScript file, move this header logic into its own file within the partials directory, like this.

Then do the same for this scrolling function.

Now we have our custom functionality all split up into separate files, let's import it.

In index.js we're asking the question, what does the home page need to function?

All pages will need these, the lazy load polyfill, cookie banner logic and the header component logic.

In fact, to save us having to import these manual for each page's file, let's create a new partial, which can be the single source of truth for these shared imports.

This way, instead of updating every page's JS file when new shared functionality is introduced, we just need to update this shared partial with the new import.

So, what else is needed on the home page? Well, it contains some Bootstrap components, so we'll need Popper and jQuery.

Instead of importing the entirety of Bootstrap, let's just import what we need, a carousel and the collapse component logic. That should save a few bytes.

We also need the scrolling function.

That should be it for the home page, now what about everything else?

Let's create the product pages JavaScript file and ask the question again, what does the products page need?

Turns out not a lot.

All it needs is the shared functionality which every page needs.

The same is true for the rest of the website's pages actually, we'd just be duplicating this file for blog and blog post.

Instead, we'll rename products.js into default.js and implement logic into the build system which says - if a JavaScript file named after the page doesn't exist, it should download this instead.

Now, we need to bring all of this together and update the build tasks.

So for each HTML page, if a JavaScript file with the same filename doesn't exist, default.js should be linked to instead.

We first need to edit the head template and change this path to a variable called js\_filename, as this is now a dynamic value.

And in the Nunjucks task in the gulp file, we'll need to declare this variable and pass it alongside the existing filename variable like so.

Here, we separate the filename variable and then use it as the default value of the js\_filename variable, which is correct, as if a JavaScript file exists which shares its name, we want to use it.

We do this check here using the native Node module FS, which is short for file system.

If a source file matching our current HTML file doesn't exist, sets the js\_filename variable to default, which will make the HTML request default.js instead.

All that's left is to pass these values to Nunjucks by returning them in an object, same as before.

Now, for example, if projects.js exists, then projects.html will request it, otherwise default.js will be requested instead, which as a bonus, will then be cached by the browser for when other barebones page require it as well.

Let's move this existing code down a little to make some space.

We need to grab all of the HTML files from the root directory and loop through them like we did with the CSS.

In here, we'll separate the filename and also create a new variable called file\_path, which we'll use to point gulp towards the page's JavaScript file in the src directory.

So for example, when the loop is at index.html, file\_path should point towards index.js.

However, as we've discussed, there isn't always going to be a JavaScript file matching the current page.

When this happens, we want both file\_path and filename to match the default.js file in the src directory.

Like in Nunjucks task, we can check if the file exists using the native FS module.

Now let's take the old code and move it below here, pointing it towards whatever our file\_path ends up being and naming it either after the current page in the loop or if required, default.js.

At the moment, we're using Bootstrap 4, which has jQuery and also Popper as dependencies.

This changes as a Bootstrap 5, which at the time of recording is still in alpha, but nonetheless available to import via NPM.

With v5, you no longer need jQuery at all, as it's all been rewritten in plain JavaScript.

And as for Popper, according to the documentation, this has only ever been required for specific Bootstrap components, none of which MLE Power is using.

Therefore, we'll upgrade to Bootstrap 5 and eliminate these dependencies.

In the command line, type this.

Now we can safely remove the jQuery import, and since none of our components use it, the Popper import as well.

On rebuild, the size of this file has dropped considerably now we've eliminated these dependencies.

It's inevitable that you'll eventually receive a request from a stakeholder to add a tracking script to the website and another request and another.

As a developer, it's your responsibility to police this as much as your role will allow for, as each of these tracking scripts incurs a performance cost.

Measure the website's performance without the script, as then a cost/benefit analysis can be run to determine if the script is worth keeping.

You can do this on a WebPageTest via the block tab, just enter the domain of the tracking script here and WebPageTest will do the rest.

The report generated will then skip the tracking script completely, giving you an idea of what to expect if it were removed.

Remember, with case studies like this, where Zalando saw a 0.7% increase in revenue when they shaved only 100 milliseconds off of their loading time, it's a conversation at least worth having.

In this instance, I've spoken with the stakeholders at MLE Power about removing the HotJar script.

It's not being used to track visitors in over six months, but I've been told it's business critical.

Here's a performance comparison showing its impact.

The biggest difference here is the Time to Interactive metric, which is nearly 75% faster without Hotjar.

This means users are able to have meaningful interactions on the website sooner, which at scale may be the difference between gaining a new customer or losing one.

Thankfully, in this case, the stakeholder agrees with my assessment, so we can safely remove Hotjar from the head template.

Unfortunately, these conversations rarely go this smoothly in real life, but it's important to have them regardless.

The biggest changes here are to Time to Interactive and Total Blocking Time, 2 JavaScript oriented performance metrics.

With Time to Interactive, it's simply a matter of having less JavaScript to delay interactivity.

Whilst with Total Blocking Time, I suspect the only remaining script which was complex enough to block the main thread was the Hotjar sript, therefore removing it has resulted in this metric going down to zero.

And then finally, less JavaScript means we have less code to download, meaning everything is fully loaded faster.

8-39 Caching Strategies

So far, we focused exclusively on the loading performance of the website's first load.

So now, for the final video in the practical segment, we'll briefly touch upon the repeat load.

One way to improve the loading performance of repeat loads is to cache our static assets more effectively.

And by effectively, I mean indefinitely.

Let's make changes so our browser only ever needs to redownload an asset if that asset has changed.

Caching behavior is server led, so to change this, we have to open the htaccess file and add this ruleset.

We first check if Apache's mod\_headers package is available.

If yes, then the contents of this block are set.

This line sets a keep alive header on requests from browser to server, this makes sure connections between the 2 remain open, reducing the time needed to serve files via reducing the number of TCP and TLS connections.

This line removes the ETag header, which stops caches and browsers from being able to validate files, which forces them to rely on the cache control policy we're defining here.

Finally, for the file extensions defined here, we give the cached files a very long max age via a cache-control header.

This value, defined in seconds, informs the browser for how long the cache should be considered fresh.

When this time has passed, then the browser would typically dump this cache and re download the asset.

Here, we've set it to a whole year.

Next, we set the assets to immutable, which indicates to the browser that these assets won't change.

So basically, with all of this, we're telling the browser that once you've cached the file, that's it, it's not going to change.

So make sure to hang on to it for as long as you're able to, up to a year.

If you open the website in Chrome's devtools and check the network tab, you can confirm these changes are working by enabling the cache control header like this.

So what happens if our files do change?

This is obviously something which is extremely likely to happen.

We'll deal with this using static asset revisioning.

Static asset revisioning means appending a hash of the file's contents to its file name, like so.

This hash is completely unique to the file, so when it changes, so too does the hash.

Therefore, as far as the browser is concerned, this is a brand new file which should be downloaded and cached.

This approach allows us to cache files indefinitely, but also download updates as they are deployed.

Install these new dependencies the usual way and import them into the gulp file.

Create a new gulp task called revision like this.

First, we grab all of the static assets we want to revise.

Here, we were on the revision module, which generates the content hash and appends it to the filename.

This module then deletes the original unhashed variants of the file as we don't want duplicates.

Our hashed files are then sent back to the www root directory.

Next, we use the rev module to generate a manifest file, which is just a JSON object of key/value pairs, which match files to their post revision file names.

This, too, is then sent to the root www directory, ready for use in the next step.

Create a new global task called rewriteRefs, the purpose of this will be to scan HTML and CSS files for references to our now revised static assets and update them to the correct paths.

The task looks like this.

First, we grab the manifest JSON generated from the previous revision task, as this will allow us to match each static asset's original filename with their new hashed filename.

We then grab the HTML and CSS files, pass them to the revRewrite module we've installed, before sending them back to their original location.

Because we want our content hashes to be as accurate as possible, this revision and rewrite process should only be run once everything else is completed.

Therefore let's export these two functions from gulp and then in our package.json file , append them here, so they run after the last build task runs.

Before we test these, as we're also applying our caching strategy to fonts, we should move them from the root www directory to the src directory.

Otherwise, as asset revisioning is applied over and over as builds are run, the filenames will grow like this, which isn't correct.

This requires another gulp task to copy these files from src to root, which we should append to our build task.

Now we've got everything in place, let's give the build a spin.

Notice how, thanks to the first task, our static assets now have hashes in their filenames, then inside the HTML and CSS, thanks to the second task, the paths are updated to the new filenames.

Here on out, if we change a file, even just a single character, then rebuild everything, the hash is now different, forcing the browser to redownload the file and discard the cache of the prior version.

Before we finish up, we need to do a quick tweak to our htaccess file. Back when we were optimizing images, we set up this rule which swaps JPGs for a WebP of the same name if it exists and the browser supports the format.

However, with hashes now being applied to file names, you're never going to have a JPG and a WebP named the same.

Therefore, we just need to tweak this regex here, so it only uses the first part when looking for a matching WebP file.

With this new caching strategy in place, on repeat views, over a much longer period than before, browsers will only spend time redownloading assets now if new versions are available.

npm install gulp-rev gulp-rev-delete-original gulp-rev-rewrite –save-dev

