Base64 Encoding & Performance, Part 1: What’s Up with Base64?

Written by **Harry Roberts** on **CSS Wizardry**.

**This is the first in a two-part post.**[**Read Part 2**](https://csswizardry.com/2017/02/base64-encoding-and-performance-part-2/)**.**

Prominent performance advice of a few years back simply stated that we should ‘reduce the number of requests we make’. Whilst this is, in general, perfectly sound advice, it’s not without its caveats. We can actually make pages load much more quickly by elegantly spreading our assets over a few more well considered requests, rather than fewer.

One of the touted ‘best practices’ born of this advice was the adoption of Base64 encoding: the act of taking an external asset (e.g. an image) and embedding it directly into the text-resource that would utilise it (e.g. a stylesheet). The upshot of which is that we reduce the number of HTTP requests, and that both assets (e.g. the stylesheet and the image) arrive at the same time. Sounds like a dream, right?

Wrong.

Unfortunately Base64 encoding assets is very much an anti-pattern[**1**](https://csswizardry.com/2017/02/base64-encoding-and-performance/?utm_source=frontendfocus&utm_medium=email#fn:1). In this article I’m hoping to share some insights as to critical path optimisation, Gzip, and of course, Base64.

Let’s Look at Some Code

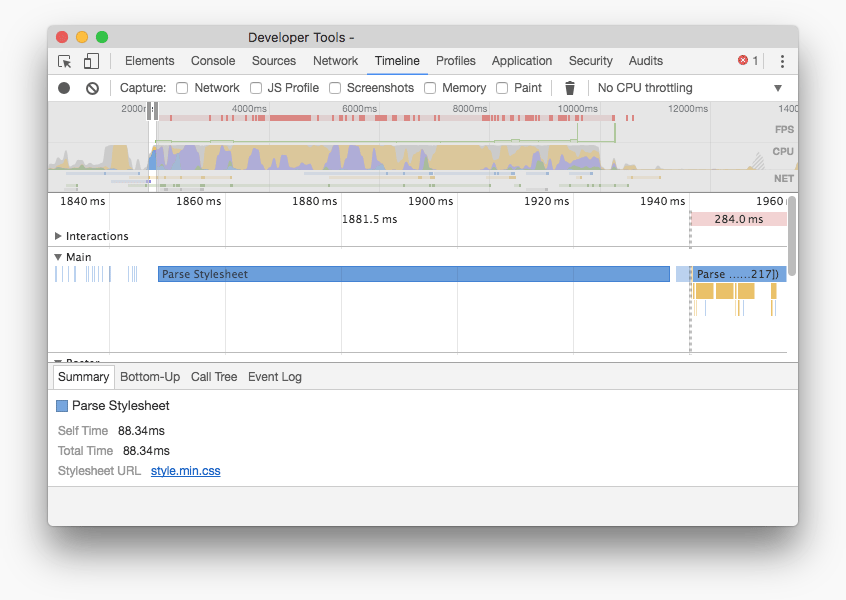
The reason I was motivated to write this article is because I’ve just been doing a [**performance audit**](https://csswizardry.com/code-reviews/) for a client, and I came across the very issues I’m about to outline. This is an actual stylesheet from an actual client: things are anonymised, but this is a completely real project.

I ran a quick network profile over the page, and discovered just one stylesheet (which is sorta good in a way because we definitely don’t want to see 12 stylesheet requests), but that stylesheet came in at a whopping **925K** after being decompressed and expanded. The actual amount of bytes coming over the wire was substantially less, but still very high at 232K.

As soon as we start seeing stylesheets of that size, we should start panicking. I was relatively certain—without even having to look—that there would be some Base64 in here. That’s not to say that I expected it to be the only factor (plugins, lack of architecture, legacy, etc. are all likely to play a part), but stylesheets that large are *usually* indicative of Base64. Still:

1. Base64 or not, 925K of CSS is terrifying.
2. Minifying it only reduces to 759K.
3. Gzipping takes us down to a mere 232K. The exact same code compressed down by 693K.
4. 232K over the wire is still terrifying.

It takes an eye watering 88ms even just to *parse* a stylesheet of that size. Getting it over the network is just the start of our troubles:



I prettified the file[**2**](https://csswizardry.com/2017/02/base64-encoding-and-performance/?utm_source=frontendfocus&utm_medium=email#fn:2), saved it to my machine, ran it through [**CSSO**](https://github.com/css/csso), then ran that minified output through Gzip on its regular setting. This is how I arrived at those numbers, and I was left looking at this:

harryroberts in ~/Sites/<client>/review/code on (master)

» csso base64.css base64.min.css

harryroberts in ~/Sites/<client>/review/code on (master)

» gzip -k base64.min.css

harryroberts in ~/Sites/<client>/review/code on (master)

» ls -lh

total 3840

-rw-r--r-- 1 harryroberts staff 925K 10 Feb 11:23 base64.css

-rw-r--r-- 1 harryroberts staff 759K 10 Feb 11:24 base64.min.css

-rw-r--r-- 1 harryroberts staff 232K 10 Feb 11:24 base64.min.css.gz

The next thing to do was see just how many of these bytes were from Base64 encoded assets. To work this out, I simply (and rather crudely) removed all lines/declarations that contained data: strings (:g/data:/d[**3**](https://csswizardry.com/2017/02/base64-encoding-and-performance/?utm_source=frontendfocus&utm_medium=email#fn:3) for the Vim users reading this). Most of this Base64 encoding was for images/sprites and a few fonts. I then saved this file as no-base64.css and ran the same minification and Gzipping over that:

harryroberts in ~/Sites/<client>/review/code on (master)

» ls -lh

total 2648

-rw-r--r-- 1 harryroberts staff 708K 10 Feb 15:54 no-base64.css

-rw-r--r-- 1 harryroberts staff 543K 10 Feb 15:54 no-base64.min.css

-rw-r--r-- 1 harryroberts staff 68K 10 Feb 15:54 no-base64.min.css.gz

In our uncompressed CSS, we’ve managed to lose a whole 217K of Base64. This still leaves us with an alarming amount of CSS (708K is pretty unwieldy), but we’ve managed to get rid of a good 23.45% of our code.

Where things really do get surprising now though is after we’ve Gzipped what’s left. We managed to go from 708K down to just 68K over the wire! That’s **a saving of 90.39%**. Wow.

Gzip Saves…

Gzip is incredible! It’s probably the single best tool for protecting users from developers. We managed to make a 90% saving over the wire just by compressing our CSS. From 708K down to 68K **for free**.

…sometimes

However, that’s Gzip working over **the non-Base64 encoded version**. If we look at the original CSS (the CSS with Base64 encoding), we find that we only made a 74.91% saving.

| **Base64?** | **Gross Size** | **Compressed Size** | **Saving** |
| --- | --- | --- | --- |
| Yes | 925K | 232K | 74.91% |
| No | 708K | 68K | 90.39% |

The difference between the two two options is a staggering 164K (70.68%). **We can send 164K less CSS over the wire just by moving those assets out to somewhere more appropriate.**

Base64 compresses terribly. Next time somebody tries the ol’ Yeah but Gzip… excuse on you, tell them about this (if they’re trying to justify Base64, that is).

So Why Is Base64 So Bad?

Okay, so we’re pretty clear now that Base64 increases filesize in a manner that Gzip can’t really help us with, but that’s only one small part of the problem. Why are we so scared of this increase in filesize? A single image might weigh well in excess of 232K, so aren’t we better off starting there?

Good question, and I’m glad you mentioned images…

We Need to Talk About Images

In order to understand how bad Base64 is, we first need to understand how *good*images are. Controversial opinion: **images aren’t as bad for performance as you think they are**.

Yes, images *are* a problem. In fact, they’re the number one contributor to page bloat. As of [**2 December 2016**](http://httparchive.org/interesting.php?a=All&l=Dec%202%202016), images make up around 1623K (or 65.46%) of the average web page. That makes our 232K stylesheet seem like a drop in the ocean by comparison. However, there are fundamental differences between how browsers treat images and stylesheets:

**Images do not block rendering; stylesheets do.**

A browser will begin to render a page regardless of whether the images have arrived or not. Heck, a browser will render a page even if images never arrive at all! Images are not critical resources, so although they do make up an inordinate amount of bytes over the wire, they are not a bottleneck.

CSS, on the other hand, *is* a critical resource. Browsers can’t begin rendering a page until they’ve constructed the render tree. Browsers can’t construct the render tree until they’ve constructed the CSSOM. They cannot construct the CSSOM until all stylesheets have arrived, been uncompressed, and parsed. CSS is a bottleneck.

Hopefully now you can see why we’re so scared of CSS bytes: they just serve to delay page rendering, and they leave the user staring at a blank screen. Hopefully, you’ve also realised the painful irony of Base64 encoding images into your CSS files: **you just turned hundreds of kilobytes of non-blocking resources into blocking ones** in the pursuit of performance. All of those images could have made their way over the network whenever they were ready, but now they’ve been forced to turn up alongside much lighter critical resources. And that doesn’t mean the images arrive sooner; it means the critical CSS arrives later. Could it really get much worse?!

Oh yes, it can.

Browsers are smart. Really smart. They make a lot of performance optimisations for us because—more often than not—they know better. Let’s think about responsive:

.masthead {

background-image: url(masthead-small.jpg);

}

@media screen and (min-width: 45em) {

.masthead {

background-image: url(masthead-medium.jpg);

}

}

@media screen and (min-width: 80em) {

.masthead {

background-image: url(masthead-large.jpg);

}

}

We’ve given the browser three potential images to use here, but it will only download one of them. It works out which one it needs, fetches that one, and leaves the other two untouched.

As soon as we Base64 these images, the bytes for all three get downloaded, effectively tripling (or thereabouts) our overhead. Here’s an actual chunk of CSS from this project (I’ve stripped out the encoded data for obvious reasons, but in full this snippet of code totalled 26K before Gzip; 18K after):

@media only screen and (-moz-min-device-pixel-ratio: 2),

only screen and (-o-min-device-pixel-ratio:2/1),

only screen and (-webkit-min-device-pixel-ratio:2),

only screen and (min-device-pixel-ratio:2),

only screen and (min-resolution:2dppx),

only screen and (min-resolution:192dpi) {

.social-icons {

background-image:url("data:image/png;base64,...");

background-size: 165px 276px;

}

.sprite.weather {

background-image: url("data:image/png;base64,...");

background-size: 260px 28px;

}

.menu-icons {

background-image: url("data:image/png;base64,...");

background-size: 200px 276px;

}

}

All users, whether on retina devices or not (heck, even users with browsers that don’t even support media queries), will be forced to download that extra 18K of CSS before their browser can even begin putting a page together for them.

Base64 encoded assets will always be downloaded, even if they’re never used. That’s wasteful at best, but when you consider that it’s waste that actually blocks rendering, it’s even worse.

And We Need to Talk About Fonts

I’ve only mentioned images so far, but fonts are almost exactly the same except for some nuance around how browsers handle the Flash Of Unstyled/Invisible Text (FOUT or FOIT). Fonts in this project total 166K of uncompressed CSS (124K Gzipped (there’s that awful compression delta again)).

Without derailing the article too much, fonts are also assets that do not naturally live on your critical path, which is great news: your page can render without them. The problem, however, is that browsers handle web fonts differently:

* **Chrome and Firefox show no text at all** for up to 3 seconds. If the web font arrives within this three seconds the text swaps from invisible to your custom font. If the font still hasn’t arrived after 3 seconds, the text swaps from invisible to whatever fallback(s) you have defined. This is FOIT.
* **IE displays the fallback font immediately** and then swaps it out for your custom font as soon as it arrives. This is FOUT. This is, in my opinion, the most elegant solution.
* **Safari shows invisible text** until the font arrives. If the font never arrives, it never shows a fallback. This is FOIT. It’s also an absolute abomination. There’s every chance that your users will never be able to see any text on your page.

In order to get around this, people started Base64 inlining their fonts into their stylesheets: if the CSS and the fonts arrive at the exact same time, then there wouldn’t be a FOIT or a FOUT, because CSSOM construction and font delivery are happening at more or less the same time.

Only, as before, moving your fonts onto the critical path doesn’t speed up their delivery, it just delays your CSS. There are some pretty intelligent font loading solutions out there, but Base64 ain’t one of them.

And We Need to Talk About Caching

Base64 also affects our ability to have more sophisticated caching strategies: by coupling our fonts, images, and styles together, they’re all governed by the same rule. This means that if we change just one hex value in our CSS somewhere—a change that might represent up to six bytes of new data—we have to redownload hundreds of kilobytes of styles, images, and fonts.

In fact, fonts are a really bad offender here: fonts are very, very unlikely to ever change. They’re a very infrequently modified resource. In fact, I just went and checked a long-running project that another client and I are working on: the last change to their CSS was yesterday; the last change to their font files was at the very beginning of the project: Initial Commit. Imagine forcing a user to redownload those fonts every single time you update anything in your stylesheet.

Base64 encoding means that we can’t cache things independently based on their rate of change, and also means that we have to cache bust unrelated things whenever something else does change. It’s a lose-lose situation.

This is basic separation of concerns: my fonts’ caching should not be dependent on my images’ caching should not be dependent on my styles’ caching.

Okay, so let’s quickly recap:

* Base64 encoding increases filesize in ways that we can’t effectively mitigate (e.g. Gzip). This increase in filesize delays rendering, because it’s happening to a render-blocking resource.
* Base64 encoding also forces non-critical assets onto the critical path. (e.g images, fonts) This means that—in this particular case—instead of needing to download 68K of CSS before we can begin rendering the page, we need to download over 3.4× that amount. We’re just keeping the user waiting for assets that they originally would have never needed to wait for!
* Base64 encoding forces all asset bytes to be downloaded, even if they’ll never be used. This is a waste of resources and, again, it’s happening on our critical path.
* Base64 encoding restricts our ability to cache assets individually; our images and fonts are now bound to the same caching rules as our styles, and vice versa.

All in all, it’s a pretty bleak situation: please avoid Base64.

Data Talks

All of this article was written using things I already know. I didn’t run any tests of have any data: it’s just how browsers work™. However, I decided to go ahead and run some tests to see just what kind of facts and figures we’re looking at. [**Head over to Part 2 to read more**](https://csswizardry.com/2017/02/base64-encoding-and-performance-part-2/).

1. There are some very exceptional cases in which it may be sensible, but you will be absolutely certain of those cases when they arise. If you’re not absolutely certain, then it probably isn’t one of those cases. Always err on the side of caution and assume that Base64 isn’t the right approach to take. [**↩**](https://csswizardry.com/2017/02/base64-encoding-and-performance/?utm_source=frontendfocus&utm_medium=email#fnref:1)
2. Open the stylesheet in Chrome’s *Sources* tab, press the {} icon at the bottom left of the file, done. [**↩**](https://csswizardry.com/2017/02/base64-encoding-and-performance/?utm_source=frontendfocus&utm_medium=email#fnref:2)
3. Run the global command (:g) across all lines; find lines that contain data: (/data:) and delete them (/d). [**↩**](https://csswizardry.com/2017/02/base64-encoding-and-performance/?utm_source=frontendfocus&utm_medium=email#fnref:3)

The Test, and Making It Fair

I started out by creating two simple HTML files that have a full-cover background image. The first was loaded normally, the second with Base64:

* [**Normal**](http://csswizardry.net/demos/base64/)
* [**Base64**](http://csswizardry.net/demos/base64/base64.html)

The [**source image**](https://www.flickr.com/photos/rockersdelight/26842162186/) was taken by my friend [**Ashley**](https://twitter.com/iamashley). I resized it down to 1440×900px, saved it out as a progressive JPEG, ran it through JPEGMini and ImageOptim, and only then did I take a Base64 encoded version:

harryroberts in ~/Sites/csswizardry.net/demos/base64 on (gh-pages)

» base64 -i masthead.jpg -o masthead.txt

This was so that the image was appropriately optimised, and that the Base64 version was as similar as we can possibly get.

I then created two stylesheets:

\* {

margin: 0;

padding: 0;

box-sizing: border-box;

}

.masthead {

height: 100vh;

background-image: url("[masthead.jpg|<data URI>]");

background-size: cover;

}

Once I had the demo files ready, I hosted them on a live URL so that that we’d be getting realistic latency and bandwidth to measure.

I opened a performance-testing-specific profile in Chrome, closed every single other tab I had open, and I was ready to begin.

I then fired open Chrome’s Timeline and began taking measurements. The process was a little like:

1. Disable caching.
2. Clear out leftover Timeline information.
3. Refresh the page and record Network and Timeline activity.
4. Completely discard any results that incurred DNS or TCP connections (I didn’t want any timings affected by unrelated network activity).
5. Take a measurement of **DOMContentLoaded**, **Load**, **First Paint**, **Parse Stylesheet**, and **Image Decode**.
6. Repeat until I got 5 sets of clean data.
7. Isolate the median of each measurement (the median is the [**correct average to take**](https://csswizardry.com/2017/01/choosing-the-correct-average/)).
8. Do the same again for the Base64 version.
9. Do it all again for Mobile (ultimately I’m collecting four sets of data: Base64 and not-Base64 on Desktop and on Mobile[**1**](https://csswizardry.com/2017/02/base64-encoding-and-performance-part-2/#fn:1)).

Point 4 was an important one: any connection activity would have skewed any results, and in an inconsistent way: I only kept results if there was absolutely zero connection overhead.

Testing Mobile

I then emulated a mid-range mobile device by throttling my CPU by 3×, and throttled the network to Regular 2G and did the whole lot again for Mobile.

You can [**see all the data**](https://docs.google.com/spreadsheets/d/1P720QU6CQ7pZUgCLtkOdUmwpwp_8nEyTMBN2zq5Ajmc/edit?usp=sharing) that I collected on Google Sheets (all numbers are in milliseconds). One thing that struck me was the quality and consistency of the data: very few statistical outliers.

Ignore the *Preloaded Image* data for now (we’ll come back to that [**later**](https://csswizardry.com/2017/02/base64-encoding-and-performance-part-2/#a-third-approach)). Desktop and Mobile data are in different sheets (see the tabs toward the bottom of the screen).

Some Insights

The data was very easy to make sense of, and it confirmed a lot of my suspicions. Feel free to look through it in more detail yourself, but I’ve extracted the most pertinent and meaningful information below:

* Expectedly, the **DOMContentLoaded event remains largely unchanged**between  the two methods on both Desktop and Mobile. There is no ‘better option’ here.
* The **Load event** across both methods is similar on Mobile, but **on Desktop Base64 is 2.02× slower** (Regular: 236ms, Base64: 476ms). Base64 is slower.
* Expectedly, **parsing stylesheets** is dramatically slower if they’re full of Base64 encoded assets. On Desktop, parsing was **over 10× slower**. On **Mobile, parsing was over 32× slower**. Base64 is eye-wateringly slower.
* On Desktop, Base64 images **decoded 1.23× faster than regular images**. Base64 is faster.
* …but on mobile, **regular images decoded 2.05× faster** than Base64 ones. Base64 is slower.
* **First Paint** is a great metric for measuring perceived performance: it tells us when the users first starts seeing something. On Desktop, regular images’ First Paint happened at 280ms, but Base64 happened at 629ms: **Base64 was 2.25× slower**.
* On **Mobile, First Paint** occurred at 774ms for regular images and at 7950ms for Base64. That’s a **10.27× slowdown for Base64**. Put another way, regular images begin painting in under 1s, whereas Base64 doesn’t start painting until almost 8s. Staggering. Base64 is drastically slower.

It’s quite clear to see that across all of these metrics, we have an outright winner: nearly everything—and on both platforms—is faster if we stay away from Base64. We need to put particular focus on lower powered devices with higher latency and restricted processing power and bandwidth, because the penalties here are substantially worse: **32× slower stylesheet parsing and 10.27× slower first paint**.

A Third Approach

One problem with loading images the regular way is the waterfall effect it has on downloads: we have to download HTML which then asks for CSS which then asks for an image, which is a very synchronous process. Base64 has the theoretical advantage in that loads the CSS and the image at the same time (in practice there is no advantage because although they both show up together, they both arrive late), which gives us a more concurrent approach to downloading assets.

Luckily, there is a way we can achieve this parallelisation without having to cram all of our images into our stylesheets. Instead of leaving the image to be a late-requested resource, we can preload it, like so:

<link rel="preload" href="masthead.jpg" as="image" />

I made another demo page:

* [**Preloaded Image**](http://csswizardry.net/demos/base64/preload.html)

By placing this tag in the head of our HTML, we can actually tell the HTML to download the image instead of leaving the CSS to ask for it later. This means that instead of having a request chain like this:

|

|-- HTML --| |

|- CSS -| |

|---------- IMAGE ----------|

|

We have one like this:

|

|-- HTML --| |

|---------- IMAGE ----------|

|- CSS -| |

|

Notice a) how much quicker we get everything completed, and b) how the image is now starting to download before the CSS file. Preloading allows us to manually promote requests for assets who normally wouldn’t get requested until some time later in the rendering of our page.

I decided to make a page that utilised a regular image, but instead of the CSS requesting it, I was going to preload it:

<link rel="preload" href="masthead.jpg" as="image" />

<title>Preloaded Image</title>

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

I didn’t notice any drastic improvements on this reduced test case because preload isn’t really useful here: I already have such a short request chain that we don’t get any real gains from reordering it. However, if we had a page with many assets, preload can certainly begin to give us some great boosts. I actually use it [**on my homepage**](https://csswizardry.com/) to [**preload the masthead**](https://github.com/csswizardry/csswizardry.github.com/blob/21044ecec9e11998d7a1e12e9f96be2aa990c652/_includes/head.html#L5-L15): this is above the fold content that is normally quite late requested, so promoting it this way does yield some significant change in perceived performance.

One very interesting thing I did notice, however, was the decode time. On Mobile, the image decoded in 25ms as opposed to Desktop’s 36.57ms.

* Preloaded images on Mobile decoded **1.46× faster than preloaded images did on Desktop**.
* Preloaded images on Mobile **decoded 3.53× faster that non-preloaded images** did on Mobile.

I’m not sure why this is happening, but if I were to make a wild guess: I would imagine images don’t get decoded until they’re actually needed, so maybe if we already have a bunch of its bytes on the device before we actually have to decode it, the process works more quickly…? Anyone reading who knows the answer to this, please tell me!

Some Interesting Things I Learned

* **Progressive JPEGs decode slower than Baseline ones.** I guess this is to be expected given how progressive JPEGs are put together, but progressive JPEGs*are* better for perceived performance. Still, it is the case that decoding a progressive JPEG takes about 3.3× as long as a baseline one. (I would still absolutely recommend using progressive, because they feel a lot faster than their baseline counterparts.)
* **Base64 images decode in one event,** whereas regular images decode across several. I’m assuming this is because a data URI can’t be decoded unless it’s complete, whereas partial JPEG data can be…?

Improving the Tests

Although I did make sure my tests were as fair and uninfluenced as possible, there are a few things I could do even better given the time (it’s the weekend, come on…):

* **Test on an actual device.** I throttled my CPU and connection using DevTools, but running these tests on an actual device would have no doubt been better.
* **Use a more suitable image on Mobile.** Because I was keeping as many variables as possible the same across tests, I used the exact same image for Desktop and Mobile. In fact, Mobile was only really simulating lowered device and network power, and was not run with smaller screens or assets. Hopefully in the real world we’d be serving a much smaller image (in terms of both dimensions and filesize) to smaller devices. I was not. I was loading the exact same files on the exact same viewport, only with hobbled connection and CPU.
* **Test a more realistic project.** Again, these were very much laboratory conditions. As I noted with preloading, this isn’t the kind of environment in which it would shine. To the same extent, expect results to be different when profiling a non-test-conditions example.

And that concludes my two-part post on the performance impact of using Base64. It kinda just feels like confirming what we already know, but it’s good to have some numbers to look at, and it’s especially important to take note of lower powered connections and devices. Base64 still feels like a huge anti-pattern.

<https://csswizardry.com/2017/02/base64-encoding-and-performance-part-1/>

<https://csswizardry.com/2017/02/base64-encoding-and-performance-part-2/>