

Heart Internet Presents

BUILD FASTER WEBSITES

A web performance guide

HeartInternet

An ebook version of this book is available at:

<https://www.heartinternet.uk/blog/build-faster-websites-free-ebook-download/>

Contents

Foreword	6
by Oliver Lindberg	
Performance Matters (and How to Convince Your Clients)	8
by Allison McKnight	
Performance Budgets: The What, Why, and How	22
by Marcos Iglesias	
You Don't Need Permission to Build Fast Websites	40
by Ryan Townsend	
Perceived Performance Matters, Too	56
by Jason Lengstorf	
The Next Four Billion: How to Make Sites Performant on Mobile Devices	68
by Jem Young	
The Critical Path – A Quest to Render Pixels Quickly	80
by Stefan Judis	

Optimise Prime: How to Optimise Images for Performance	98
Henri Helvetica	
Make Your Animations Perform Well	118
by Anna Migas	
Performant Web Font Techniques	138
by Matt James	
Measuring Performance	152
by Andy Davies	

FOREWORD



The editor

Oliver Lindberg

Oliver Lindberg is an independent editor, content consultant and founder of new web events series **Pixel Pioneers** (<https://pixelpioneers.co>), based in Bath, England. Formerly the editor of .net magazine, he's been involved with the web design and development industry for more than a decade and helps businesses across the world create content that connects with their customers.

In 2017, the size of the average web page reached 3MB. The modern web is becoming more and more bloated, and page sizes continue to grow just as steadily as the number of devices we're designing for. The need to optimise our sites and apps for performance has become more important than ever, especially as the next billion people are getting ready to come online, most of them on mobile in emerging markets.

In this book — written by front-end developers, engineers, and web performance advocates from the UK, US, and Europe — we'll explain just why performance matters and how to convince your clients. We'll cover how to set up performance budgets, how to build a fast site even if you don't get permission from your client, and how to subtly trick users into thinking a site loads faster than it actually does.

There are many things you can do to improve the performance of your site, and in this book we'll present a range of techniques covering images, animations, and web fonts. We'll also look at how to make sites performant on mobile devices in particular, optimise JavaScript and CSS (which can block the rendering of a web page) and how to measure the performance of your sites. Whether you want to instigate change within your organisation or learn about the latest best practices, you'll find something for you here.

Some of our authors work for big companies like Netflix, Etsy, Eventbrite, and IBM. Others are freelance or employed by software development agencies and start-ups. What they all have in common is a passion for building better and faster websites and ultimately creating an enjoyable user experience for all.

PERFORMANCE MATTERS

(AND HOW TO CONVINCE YOUR CLIENTS)



The author
Allison McKnight

Allison McKnight (twitter.com/aemcknig) is a software engineer at Etsy. During her four years on the performance team, she focused on creating tools that allowed teams to see how their changes affected site performance, understand how those changes impacted the user experience, and feel empowered to make product decisions based on performance changes.

If you've ever used a slow website (and chances are you have), you know that performance is an important part of the user experience.

But performance isn't always the focus of teams developing new sites or building out existing features. Too often, performance is seen as a "bonus" feature rather than a key part of the user experience. Stakeholders urge developers to build new features that they hope will drive key business metrics, such as gross merchandise sales (GMS) or user engagement, leaving developers little time to focus on performance. What clients and stakeholders might not realise is that performance *is* one of those features.

In fact, performance is an *essential part* of the user experience. Users who experience a slow website are seeing a degraded experience, and they are less likely to interact with, purchase from, or return to that site as a result. In turn, this decreased user engagement impacts business metrics that stakeholders and clients care about.

Without a solid understanding of the link between performance and business metrics, however, it's hard to shift stakeholders' focus from building sleek new features to optimising the current product.

This is where you come in. You can deliver the best product for your clients, users, and stakeholders by understanding:

- How performance affects the user experience
- How the impact of performance on the user experience influences the metrics that matter to your clients and stakeholders
- How you can share this information with clients and stakeholders in a way that shows the importance of focusing on their products' performance

Performance is user experience

Many of us understand that trying to use a sluggish website can be frustrating, likely through firsthand experience. If you've ever become fed up with a slow experience, you're not alone — a 2016 study by Google's Doubleclick discovered that "53% of mobile site visits are abandoned if pages take longer than 3 seconds to load" [1]. Even on mobile, where users are on slower hardware and likely poor connections, the bar is high.

But the impact that performance can have on the user experience is much deeper than you might imagine.

While a long page load time will make a website feel slow, it can do even more damage to users' perception of your site. A study performed by Radware in 2013 [2] found that when one website was artificially slowed down by 500 milliseconds, visitors' perception of the brand changed. When interviewed, visitors who experienced the site at a normal speed generally had more positive than negative opinions and praised the usability and friendliness of the site.

But users who experienced the slower version of the site shared different feedback: beyond complaining that the site was slow, they described their experiences as frustrating, complicated, inelegant, boring, childlike, and tacky. Tacky! Although the only difference in these two groups' experience was the site's load time, users in the slower group had a completely different perception of the website and its brand.

In fact, your site's performance influences whether users get to experience the site at all. Consider emerging markets, where infrastructure constraints can multiply the impact of poor performance.

YouTube learned about the importance of performance in these markets in 2012, when they sought to optimise their video page [3]. Realising that YouTube's video page was ridiculously large, one engineer set out to build an optimised, lightweight version of the page. Engineers expected page load times for the optimised page to be much shorter than load times for the bulky original version of the page. But the data was surprising: load times were actually significantly *longer* for the optimised version of the page — two minutes on average!

This was perplexing at first, but when engineers looked into the data, they discovered that the lightweight version of the page was getting a disproportionate number of views from areas with very low connectivity. For users in these areas, the older, unoptimised version of the page had taken *20 minutes* to load. With the new lightweight version of the page, these users were finally able to load and watch a video on YouTube in a reasonable amount of time. By optimising the video page, YouTube was able to reach entire regions of users that had previously been alienated by long page load times.

Performance is an important part of the user's experience. Its impact ranges from frustration with a slow experience to a user's willingness or even ability to use a site. In order to provide users with a great experience, we need to focus on our sites' performance.

Making the case for performance

Sometimes, however, knowing that performance affects the user experience is not enough. Clients and stakeholders who are focused on running a successful business have their own set of business metrics and goals, and it isn't always obvious to them that performance should be one of them. Furthermore, it can be hard to understand the true performance of one's own website — loading a site on a computer in an office with a good, fast internet connection might make the site seem to load quickly, but users who are loading the site at home, on the train, or on their mobile device might have a different experience.

In order to understand the importance of performance, stakeholders need to first understand that performance impacts the business metrics that they care about and identify that there is room for performance improvement in their product.

The following section highlights strategies and tools that you can use to show clients and stakeholders that performance is relevant to their business. By making performance relevant, you can make the case for focusing on performance.

Link performance to business goals

A first step towards convincing stakeholders that spending time on performance optimisation will help move their business forward is to link performance to the metrics and initiatives that they care about.

Performance is a business optimisation

While clients do care about the user experience, they have good reason to focus on business metrics and projects that will move those metrics. After all, they are running a business: they need to maintain and grow their key metrics so that they can deliver value to their own stakeholders.

Focusing on these business metrics does not preclude performance work. In fact, when your goals include improving business metrics, performance becomes even more important. When users experience slowness on a website, they are less likely to engage with the site, and that reluctance impacts business metrics.

It has been demonstrated time and again that sluggish load times lead to sluggish business metrics and that improving load times can lead to improved business metrics. Here are just a few examples:

- By implementing a server-side performance improvement [4] that brought average response time down from 4.80 seconds to 3.95 seconds, COOK improved conversion rate by 7% (from 3.82% to 4.09%).
- An analysis by Google DoubleClick [1] found that pages that load in 5 seconds or less on mobile generate up to twice as much ad revenue as sites that take 19 seconds to load. (At the time of the study, 19 seconds was the average load time for mobile sites on a 3G connection.)

- When adding more results to Google's search page increased page load time by 0.5 seconds, searches by users decreased by 20% [5].
- When AliExpress reduced page load time by 36%, the number of orders placed on the site increased by 10.5% [6].
- When the Financial Times artificially added a one-second delay to their page load time, users viewed 4.9% fewer articles on average [7].
- During Obama's 2012 campaign for the United States presidency, improved performance on the campaign's donation page led to a 14% increase in donation conversions [8].

Focus on your stakeholders' key metrics and business goals

Clearly, focusing on performance has worked out well for these companies. But business goals (and the metrics that track progress towards those goals) vary widely from company to company. You need to make performance relevant to your stakeholders' goals by showing how it can impact the key metrics that they care about.

Ask yourself which metrics your stakeholders are focused on improving. Perhaps they want to increase the number of new or repeat visits to the site. If your client publishes subscription media, they may aim to optimise the number of subscriptions to their service. Social media sites may aim to improve user engagement by focusing on favourite or comment rates.

Find the metric that your clients are focused on improving. Then, tailor your argument to your stakeholders by showing them how

performance optimisations can influence that particular metric. Once your client sees performance as a lever that can move exactly the metrics that they seek to improve, performance will no longer be just a “nice-to-have” — it will become a way for your team to achieve its goals.

A valuable tool in this endeavor is WPO Stats [9], which collects and lists dozens of case studies that link performance to metrics such as conversion, engagement, bounce rate, page views, and more. Using this tool, you can find case studies that relate performance to the business metrics that stakeholders care about.

Short-term initiatives

In addition to long-term goals, such as optimising a set of business metrics, your company likely has current initiatives that will help it make progress towards those goals. This is another opportunity for you to make performance relevant to your clients. Performance can be used to make progress on the short-term initiatives that your client hopes will help them make progress towards their overarching goals.

For instance, at a time when the company was trying to improve the discoverability of their site and product, Smart Furniture did some research to discover that Google considers page load time in its ranking of search results. The team was able to improve the performance of their site [10], resulting in a higher placement in Google’s search results and leading to more visitors to the site from Google searches.

Here, the team at Smart Furniture realised that performance could be used to accomplish a short-term initiative (to push Smart Furniture results higher in Google search results) that supported a long-term business metric (visits to Smart Furniture).

By considering short-term initiatives that your clients are taking on, you can show how performance can help your clients reach their goals. At the same time, you're getting performance "in the door" so that your stakeholders will consider performance when thinking about future initiatives to support their business goals.

Show, don't tell

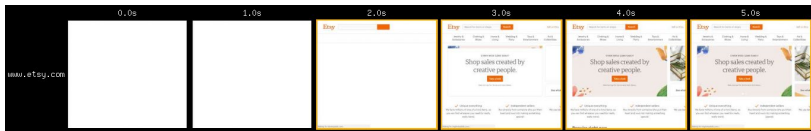
Another barrier to getting buy-in from your clients and stakeholders is that it is often hard to empathise with users who experience slowness on your site.

Because offices tend to offer good internet connections, and because website development usually happens on computers rather than on slower mobile devices, many of us are used to the pages we're working on to load relatively quickly. This makes it easy to forget that users in other areas or on different devices might experience much slower page load times.

To help stakeholders understand what users are experiencing on their site, you can *show* them how their site loads for users on different devices or in different parts of the world.

WebPagetest [11] is a tool that does exactly that. It will help you demonstrate to your stakeholders how real users experience their site. WebPagetest can load a page and generate a video of the page loading as well as a filmstrip view showing the visual progress of the page load. Showing a video of your page loading on a mobile device, or on a sub-optimal network, can help your clients

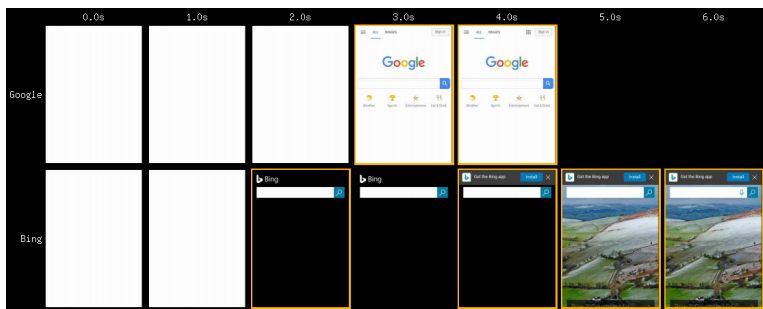
and stakeholders *empathise* with their users, understand how their site performs in the wild, and appreciate the need for a focus on performance.



WebPagetest's filmstrip view shows what users experience as they load the Etsy homepage

Compare to the competition

This technique can also be used to capitalise on your client's competitive spirit by highlighting where your product stands in relation to its competition. WebPagetest's comparison view allows you to load two pages — for instance, your site and a competing site — and show how the load times differ. If your client's site has a competitor, use this tool to show stakeholders how their site stacks up.



WebPagetest's filmstrip view comparing the page load experiences of Google and Bing

Maybe your site is faster than the competition — in this case, the comparison should demonstrate to stakeholders that performance is an advantage that your product has over the competition, and that it should be maintained. If your site is neck-and-neck with the competition, or sluggish in comparison, it's a good time to focus on performance optimisations — the resulting improvement in the user experience will help you win customers away from your competition.

Make performance relevant

Performance is an essential part of the user experience, and because you strive to provide the best experience for your users, performance is important and relevant to you. You can help your clients see the importance of performance by making performance relevant to *them*: by showing them the impact of performance on the user experience, users' perception of their product, and the business metrics that stakeholders care about. By making performance relevant to your client's goals and priorities, you can show them that focusing on performance will bring them closer to achieving those goals.

Resources

[1] Alex Shellhammer, DoubleClick

The need for mobile speed: How mobile latency impacts publisher revenue
<https://www.doubleclickbygoogle.com/articles/mobile-speed-matters/>

[2] Tammy Everts, Radware Blog

Mobile Web Stress: The impact of network speed on emotional engagement and brand perception
<https://blog.radware.com/applicationdelivery/applicationaccelerationoptimization/2013/12/mobile-web-stress-the-impact-of-network-speed-on-emotional-engagement-and-brand-perception-report/>

[3] Chris Zacharias

Page Weight Matters
<http://blog.chriszacharias.com/page-weight-matters>

[4] NCC Group

COOK Case Study
<https://www.nccgroup.trust/uk/about-us/resources/cook-real-user-monitoring-case-study/?style=Website+Performance&resources=Case+Studies>

[5] Greg Linden

Marissa Mayer at Web 2.0
<https://glinden.blogspot.co.uk/2006/11/marissa-mayer-at-web-20.html>

[6] Dr. Dongbai Guo, Akamai Edge Conference

Future of Commerce
<https://edge.akamai.com/ec/us/highlights/keynote-speakers.jsp>

[7] Matt Chadburn & Gadi Lahav, Financial Times' Engine Room

A faster FT.com
<http://engineroom.ft.com/2016/04/04/a-faster-ft-com/>

[8] Kyle Rush

Meet the Obama campaign's \$250 million fundraising platform
<http://kylerush.net/blog/meet-the-obama-campaigns-250-million-fundraising-platform/>

[9] WPO Stats

<https://wpostats.com/>

[10] Akamai

Akamai Accelerates Smartfurniture.com, Resulting In Higher Search Engine Rankings That Have Led To More Sales And Revenues

<https://www.akamai.com/uk/en/about/news/press/2010-press/akamai-accelerates-smartfurniture-com-resulting-in-higher-search-engine-rankings-that-have-led-to-more-sales-and-revenues.jsp>

[11] WebPagetest

<https://www.webpagetest.org/>

PERFORMANCE BUDGETS: THE WHAT, WHY, AND HOW



The author
Marcos Iglesias

Marcos Iglesias (www.marcosiglesias.com) is a senior software engineer who builds compelling user interfaces at Eventbrite. Marcos is passionate about improving developer efficiency, building tools, and setting up processes to save his peers both time and effort. On top of all that, he enjoys contributing to Eventbrite's Engineering blog (www.eventbrite.com/engineering), giving talks, and maintaining Britecharts (eventbrite.github.io/britecharts), the open source charting library. Marcos has eight years of experience developing web applications and sites with front-end technologies in different fields, including e-commerce, online banking, and SaaS platforms.

Whether you are a lonely freelancer, work in a modest agency, or a large-scale product company, performance budgets can help you get the web performance conversation started. They will help you sell web performance to your clients, and allow you to make quick decisions with a clear reference point.

Arguing between design, product, and engineering about every feature that needs to be built is hard. There are many stakeholders and considerations when creating new features or products. This process can go wrong, which results in delays, feature creep, frustration, and ultimately, inferior products for our users.

Definition

A performance budget is a set of values for web performance metrics that — agreed by engineering, product, and design — defines the minimum web experience that your users will enjoy.

It acts as a reference for making decisions about feature inclusion, update, removal or deprecation.

Why use performance budgets?

The short answer is **users**. They won't have to close your site because they lack the patience to wait for it to load. But not only users will benefit.

For **web engineers**, tracking performance budgets will be helpful when comparing different approaches to a problem. They will contribute to highlight the weight, processing, and code parsing time that different frameworks, vendors or solutions will add. Performance budgets are a tool for translating web performance into visible business value.

For **designers**, a performance budget will show the limitations of less effective designs. Knowing the boundaries will help **designers** find a better, more performance-friendly solution.

For **product managers**, performance budgets help to prioritise features and to understand how they contribute to the final user experience. They will have a measure to push for improvement campaigns, mapping them to fewer bounce rates and more conversions.

And finally, for **managers**, performance budgets will help to track the progress of improvements by the team.

How to use performance budgets

Your team will measure every decision against the performance budgets. Product and design will try not to surpass the budgets, and engineering will make sure they don't get surpassed and work actively to decrease them even more.

When something overcomes the threshold, as Steve Souders — from SpeedCurve, and formerly Google and Yahoo — advises [1], you can either:

- Optimise an existing feature or asset on the page
- Remove an existing feature or asset from the page
- Don't add the new feature or asset

Brutal and straightforward, isn't it?

On a daily basis, tracking performance budgets will help your design and development teams to work with web performance in mind. For example, the group decides to utilise an SVG instead of

a picture or chooses a lighter weight JavaScript framework instead of a weightier, more fashionable counterpart.

Setting performance budgets

A factor to consider when setting a budget is the type of hosted content. Landing pages, marketing brochures, homepages, product listings, video content and web applications have all different use cases and require different strategies. You will want to set a performance budget for each of these types of content.

Another consideration is your team and their motivation. Will they be excited about beating the competition? Maybe the team is more driven by visible improvements against the current version or just 'doing it right' following industry guidelines. In this section, you will learn about the various approaches and alternative strategies.

Be faster than your competitors

This is one of the most common approaches. You can create a performance budget by running your competitors sites through WebPagetest [2], and choose [3] metrics [4] so you can beat them.

In case you are wondering what it means to 'beat them', there is a quantitative way to measure it, using the 20% Rule [5].

This rule states that, for our users to be able to perceive a noticeable improvement between more performant and less performant experiences, the difference in speed should be no less than 20 per cent. Below that point, our users will think both sites are more or less fast.

Coming back to the competitor's comparison, we can say this is an easy way of approaching performance budgets if you

need the sign-off from your clients or management. It's easy to conceptualise and will also be a compelling strategy to sell to your team and managers. After all, who doesn't want to beat 'that other' company?

It will probably need a system that keeps track of your current and emerging competitors, so that your performance budgets move accordingly.

Improve the current version

This is an excellent approach if you are developing a redesign or code rewrite. You could do a performance audit on the different page types on your site and create your new performance budget by reducing those metrics by 20 per cent.

Follow academic or industry guidelines

Perhaps this approach is the best one when you have an entirely new product that you are building from scratch. This strategy will likely need some extra feedback cycles, as its initial values will be guesses to be contrasted with the experience of real users on a wide range of devices and network conditions.

You could try to follow Human-Computer Interaction (HCI) guidelines based on user limits in 'Response Times' [6], published by Jakob Nielsen in his book Usability Engineering [7]. More recently, we have seen Google's Chrome Team expose a model for web performance they called 'RAIL' [8] based on setting goals for the different 'actions' web users perform within web applications.

Optimise business value

This is a highly focused way of setting performance budgets. The vast amount of research and case studies shared by tech companies about their performance improvement campaigns show the economic benefits they produced. You can refer to Google's [9] and Yahoo's [10] cases among others.

First, identify the most critical parts of your site; they could be a landing page, or maybe the checkout flow. Set performance budgets on those key sections and try to improve them by the 20% rule. Release it to the user, evaluate, and repeat.

You can set performance budgets using one or a combination of strategies. It will all depend on what makes sense to you as a company or to your clients. Don't worry if your first budgets are off or not representative. At the end of the day it's an estimation, and we know how flawed we humans are at that task.

Metrics

Measuring web performance is hard. Since the very beginning of web performance this has been a source for endless discussions: Should you favour fewer HTTP requests and heavier assets? One big cacheable JavaScript bundle or on-demand module downloads?

Tester: VM2-02-192.168.10.70

First View only

Test runs: 3

[Re-run the test](#)

[Raw page data](#) - [Raw object data](#)

[Export HTTP Archive \(.har\)](#)

[View Test Log](#)

Performance Results (Median Run)

	Load Time	First Byte	Start Render	Speed Index	Document Complete			Fully Loaded			
					Time	Requests	Bytes In	Time	Requests	Bytes In	Cost
First View (Run 3)	3.310s	1.248s	3.608s	3631	3.310s	9	299 KB	3.671s	14	319 KB	\$----

[Plot Full Results](#)

WebPagetest results showing several metrics

In today's performance world, this is still on-going. Here are some metrics you could use, ordered in groups, based on the work of Tim Kadlec [11].

Quantity-based metrics

Based on raw values, these are conceptually the easiest. Here we could include the number of HTTP requests made, the weight of the images, styles and JavaScript assets and even the HTML document weight. These metrics describe the 'browser experience' and are not directly related to the user experience.

I like to use these metrics to create lapidary sentences like 'we are downloading 2MB of images!' They are excellent for raising alarms, but usually far from what I would consider scientific truths. When and how you download those assets or make those requests is more important than the total sum of them.

Rule-based metrics

Less fashionable lately, rule-based scores are generated by web applications and browser extensions, and their output is a single or a series of rates to your site or properties. The earliest example in this category is Yahoo's YSlow [12], followed by Google's PageSpeed Insights [13] and more lately Lighthouse [14]. My favorite, WebPagetest, also includes a grading system.

Web Page Performance Test for

www.marcosiglesias.com

From: Dulles, VA - Chrome - Cable
11/28/2017, 10:14:29 PM



WebPagetest results including scores

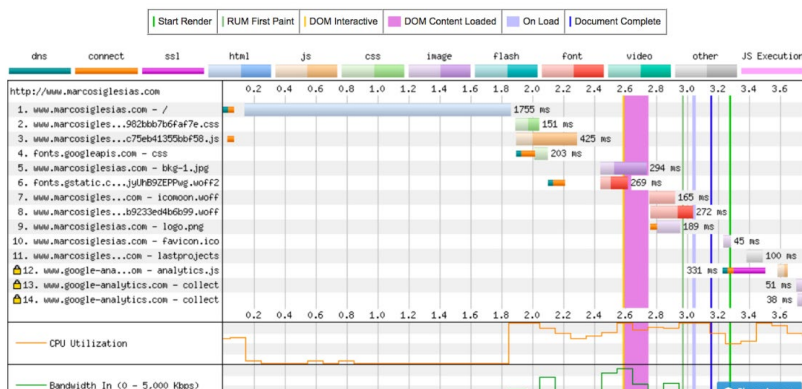
A good thing about these metrics is that they focus on scores, which are easy to communicate and measure. They also provide 'action items' to improve the performance of the tested page. Sadly, they also measure the browser experience, so even if they include more metrics to give an 'overall' view of the web performance, they are not centered on user experience.

You could probably make an exception here for Lighthouse, as it tracks some user-centric metrics like 'Time to Interactive' [15] and 'Perceptual Speed Index,' measured by Google Chrome.

Milestone timings

These are usually time-based metrics reported by the browsers. You can find them in the 'Network' tab of the browser's developer tools. 'Page Load,' 'DOMContentLoaded' or 'Time to First Byte' are good examples.

Waterfall View



The waterfall view on WebPagetest shows the more important milestones

'Time to First Byte' gives you a good measure of your backend speed, and the new 'Start Render' milestone tells you when the page started its render. These, along with the 'Page Load' event, describe the user experience.

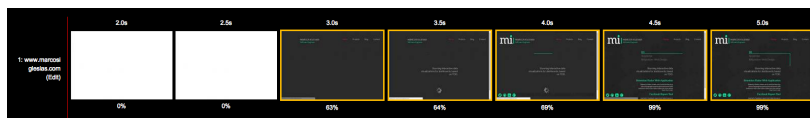
They tell you when milestones trigger, but that's not the whole picture. What the users see inbetween 'Start Render' and 'Page Load' is not recorded here. That's where the Speed Index comes in.

Speed Index

Pat Meenan created Speed Index as part of WebPagetest to measure how the page will show its contents to the user. From the documentation:

“The Speed Index is the average time at which visible parts of the page are displayed.” [16]

This measurement describes the user experience of the site better, so much that it’s dependent on the viewport of the users. When measuring the Speed Index, you will discover how a user with any given browser, viewport size, and network connection will experience the load of the contents of your site.



WebPagetest analyses the visual progress of the webpage

It’s an excellent metric but still doesn’t give us a complete picture. Imagine that you have a web application that quickly loads its content, but later, it spends a lot of time downloading JavaScript bundles. It will need to parse those files to start the application. All this time, the users won’t be able to interact with the page. That’s where custom metrics shine.

Custom metrics

Custom metrics are events that you will trigger when something interesting happens.

You could track a hero image's render time, as described by Steve Souders [17]. Or it could be the main action of the application, like the well-known example of the 'Time to First Tweet' [18].

In modern development, you can use the Web Performance API [19] and get custom metrics tracked by the browser and reported to your servers.

They have one big drawback: they are not scalable. Doing this for all the actions of our site would be absurd. However, you could do it by 'templates,' targeting different content types with generic 'template-based' custom metrics.

Custom metrics also require some testing and adjusting, and the techniques to measure are sometimes tricky. They are the more granular metric.

Choosing metrics and thresholds

Favour metrics that show you information about the experience of your users. Optimise when they see the content and when they can interact with it.

Choose threshold values that push you to keep on creating business value by improving your product. Use the conversions and analytics data to establish a feedback loop. Remember, the performance budgets should not be carved in stone, and they could change, along with the business goals.

Contrast it with lots of feedback, and check it on the most common devices for your audience. There will be massive differences between iPhone-heavy users and limited feature-phones.

Network conditions could also be an essential element. Is your audience checking the site on the train during commuting? Do they always check your dashboards from their work's desktop computer?

In the end, your performance budgets will be created taking into account your type of content, the metrics you want (or you can track), and your research of audience, devices and network conditions.

Examples of performance budgets

Some specific examples include the performance budget of BBC News [20], which was to make the page usable in less than 10 seconds under a GPRS connection in 2012.

In 2014 Smashing Magazine mentioned a performance budget optimised for the Critical Rendering Path [21], so all CSS, JavaScript, and HTML for the 'Above the Fold' area of the site would need to fit into the first 14Kb. They also added a challenging goal for the Speed Index: less than 1,000 milliseconds.

Recently, Alex Russell from Google, mentioned a performance budget based on Time To Interactive [22]: TTI under 5 seconds for the first load and under 2 seconds for the repeating visitor.

I would recommend you start picking a 'Start Render', a 'Speed Index' and a set of custom metrics that represents business value for your company or client. Get the ball rolling and review it soon.

Tracking performance budgets

Choosing between tools will depend on the priority of web performance for your client or company, the budget you are allowed to spend and the scale of the site. There are many tools in the market, some of them are free or do-it-yourself, while others are more aimed at enterprises.

Free options

- **Sitespeed.io** [23] enables you to set up a whole infrastructure of testing for your site. It offers complete reports and can now keep track of the Speed Index and custom metrics by using a private instance of WebPagetest.
- **SpeedTracker** [24] is a more lightweight solution that tracks the Speed Index and sends you alerts. It will work with a free WebPagetest account key, although it will be limited to the number of checks per day.

Subscription-based tools

- **Calibre [25]** adds to the mix some of the Chrome-specific metrics like Time To Interactive that are provided by Lighthouse.
- **Rigor [26]** is a pretty complete solution, although at the time of writing this hasn't added Speed Index as a metric.
- **SpeedCurve [27]** is by far my favourite of the paid solutions. It allows Speed Index and custom metrics and provides a lot of useful data and comparisons within their dashboards.

Complete custom solutions will take time and resources from your team; this will be a good investment if you value performance considerably. If you are more interested in lower budgets, and try to start small to show the business value before going further, then SpeedTracker, Calibre or Rigor could be great options.

When dealing with high traffic sites, SpeedCurve and Rigor seem to be good choices, with SpeedCurve being the absolute best of the bunch if you can afford it. For your side projects or hobbies, I would recommend SpeedTracker or Sitespeed.io.

Conclusion

Performance budgets are self-imposed limits to web performance, which — employed by the team as a reference — help them create an enjoyable user experience. It's critical creating performance budgets early in the project lifespan, as the effects will guide the development.

They serve all the components of a web development team, forcing them to take performance into account. Several estimating strategies and metrics allow your budgets to adapt to your content, audience and technical limits. You can choose between free and paid tools to help you track them.

Adopting performance budgets is a workflow change that influences product, design, and engineering, and it all happens with the end user in mind.

Like other transformations, it will take some time to figure out, and some flexibility will be required. It's a challenge that you can overcome by using the data to show progress, business value and customer satisfaction.

Resources

[1] Tim Kadlec

Setting a Performance Budget

<https://timkadlec.com/2013/01/setting-a-performance-budget/>

[2] WebPagetest

<http://www.webpagetest.org/>

[3] The Path to Performance podcast

Jeff Lembeck of Filament Group

<https://pathtoperf.com/2015/05/05/03-with-jeff-lembeck.html>

[4] Responsive Web Design podcast

Niko Vijayaratnam & John Cleveley of BBC News

<https://responsivewebdesign.com/podcast/bbc/>

[5] Denys Mishunov, Smashing Magazine

The Need for Performance Optimization: The 20% Rule

<https://www.smashingmagazine.com/2015/09/why-performance-matters-the-perception-of-time/#the-need-for-performance-optimization-the-20-rule>

[6] Jakob Nielsen, Nielsen Norman Group

Response Times: The 3 Important Limits

<https://www.nngroup.com/articles/response-times-3-important-limits/>

[7] Jakob Nielsen, Nielsen Norman Group

Usability Engineering

<https://www.nngroup.com/books/usability-engineering/>

[8] Paul Irish, Smashing Magazine

Introducing RAIL: A User-Centric Model for Performance

<https://www.smashingmagazine.com/2015/10/rail-user-centric-model-performance/>

[9] Marissa Mayer, Google

In Search of... A better, faster, stronger Web

<http://assets.en.oreilly.com/1/event/29/Keynote%20Presentation%202.pdf>

[10] Stoyan Stefanov, Yahoo

Don't make me wait! Or Building High-Performance Web Applications

<https://www.slideshare.net/stoyan/dont-make-me-wait-or-building-highperformance-web-applications>

[11] Tim Kadlec

Performance Budget Metrics

<https://timkadlec.com/2014/11/performance-budget-metrics/>

[12] YSlow

<http://yslow.org/>

[13] PageSpeed Insights

<https://developers.google.com/speed/pagespeed/insights/>

[14] Lighthouse

<https://developers.google.com/web/tools/lighthouse/>

[15] Time To Interactive

<https://github.com/WPO-Foundation/webpagetest/blob/master/docs/Metrics/TimeToInteractive.md>

[16] WebPagetest

Documentation – Speed Index

<https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index>

[17] Steve Souders

Hero Image Custom Metrics

<https://www.stevesouders.com/blog/2015/05/12/hero-image-custom-metrics/>

[18] Twitter

Improving performance on twitter.com

https://blog.twitter.com/engineering/en_us/a/2012/improving-performance-on-twittercom.html

[19] Mozilla

Web Performance API

<https://developer.mozilla.org/en-US/docs/Web/API/Performance>

[20] Tom Maslen

The story of how BBC News fell in love with Responsive Web Design

<https://speakerdeck.com/tmaslen/moving-swiftly-the-story-of-how-bbc-news-fell-in-love-with-responsive-web-design>

[21] Vitaly Friedman, Smashing Magazine

Improving Smashing Magazine's Web Performance: A Case Study

<https://www.smashingmagazine.com/2014/09/improving-smashing-magazine-performance-case-study/>

[22] Alex Russell

Can You Afford It? Real-world Web Performance Budgets

<https://infrequently.org/2017/10/can-you-afford-it-real-world-web-performance-budgets/>

[23] Sitespeed.io

<https://www.sitespeed.io/>

[24] SpeedTracker

<https://speedtracker.org/>

[25] Calibre

<https://calibreapp.com/>

[26] Rigor

<https://rigor.com/>

[27] SpeedCurve

<https://speedcurve.com/>

YOU DON'T NEED PERMISSION TO BUILD FAST WEBSITES



The author
Ryan Townsend

With over 15 years' experience developing for the web and an undying passion for web performance, Ryan Townsend (twitter.com/RyanTownsend) is the CTO of Shift Commerce – a SaaS e-commerce platform bringing agility to mid-to-enterprise retailers. His pragmatic and performance-led outlook means that – on the rare occasion that he does wear a shirt – his sleeves stay firmly rolled up: even as an exec, his favourite place is right there in the thick of things with his team. Outside of the office, you'll usually find him picking up heavy objects in the gym or falling off his mountain bike down the side of a hill.

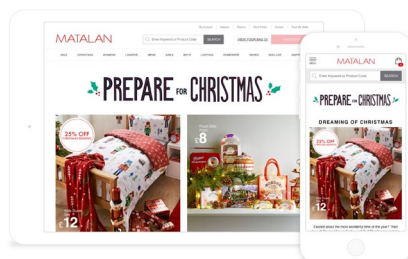
British fashion and homeware retailer Matalan is really forward-thinking when it comes to the web, and performance is something in which they place considerable value. Likewise, my team at Shift Commerce and I get a kick out of building websites that load in a blink of an eye — for many of us, it's a matter of pride.

So, ultimately, when Matalan approached us to implement a new platform, my team built a fast site. Business as usual.

Of course, your mileage may vary with this approach. As there may be commercial sacrifices, or a change to the user experience, you may need permission to make the *fastest* website, but you certainly don't need it to make a fast one.

The wonderful thing is that the tools we have at our disposal are getting better all the time. Thanks to the herculean efforts of the browser vendors, visibility into how browsers operate is increasing, our ability to resolve issues natively without resorting to hacks is getting easier, and there is simply more we can do without requiring costly rewrites or introducing new technology.

Below are some of the techniques implemented behind-the-scenes by my team to not only avoid performance issues, but to deliver improvements as part of Matalan's relaunch too.



Matalan's website on desktop and mobile – responsive imagery ensures high performance

Respect the network

If there's one takeaway I can give you from this entire article, it's to treat the network as a precious and finite resource.

It doesn't matter whether your customers are based in developed countries, with super-fast 4G connectivity and the latest powerful smartphones — high latency and low bandwidth can occur for a whole variety of reasons — anyone who's ridden a train in the UK will know this! So, we should be treading very carefully whenever making network requests — the last thing any business wants is reasons for their customers to go elsewhere.

If you need any further convincing, see Harry Roberts' article on The Fallacies of Distributed Computing [1], a concept penned way back in 1994.

Limit critical-path assets

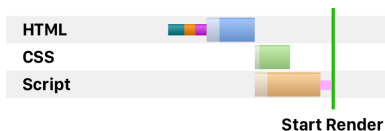
You may have heard the phrase “the fastest request is one never made” — minimising the number of requests required for a meaningful paint to occur will deliver significant gains for the user experience, even if we're merely moving them off the critical rendering path but not removing them altogether.

Typically the requests we're looking for are any stylesheet `<link>` tags or non-asynchronous/deferred `<script>` tags either in the `<head>` or anywhere above the bottom of `<body>`, this also extends to any CSS imports or web fonts loaded within stylesheets. Note: this includes both first and third party requests.

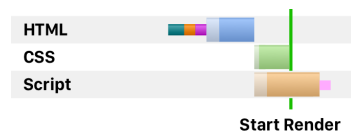
Every single one of these requests is a Single Point of Failure (SPOF), and any latency or slowness in loading them will directly add time to the visitor being sat staring at a blank white screen.

When we are starting from scratch and start adding each of these tags, we should consider whether:

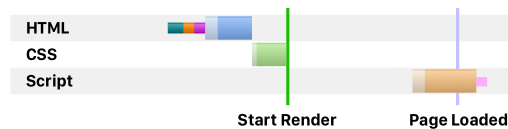
1. the tag needs to be synchronous — if not, make it asynchronous or deferred to page load.
2. the contents need to be entirely synchronous, or whether you could break out into two separate sync/async bundles, with the most minimal code being synchronous.
3. you could (or should) inline it within the HTML itself to avoid those extra round trips with the server.



Synchronous Tag



Asynchronous Tag



Deferred Tag

Google AMP is quite a controversial project amongst the web development community, but whatever your views are, many of their requirements around implementation make excellent guidelines for non-AMP projects too:

1. No network requests allowed on the critical rendering path
2. Up to 48kb CSS, delivered via inline `<style>` tag only
3. No CSS `@import`
4. Minimal options for custom web fonts (limited to just four whitelisted third parties)

Controlling the network

If you must hit the network for critical path resources, protecting the requests we make is vital.

Thankfully, with the recent introduction of Service Workers, we can intercept network requests and modify how they are handled.

There are many different strategies we could adopt:

- enforcing a short timeout on third party critical-path requests
- serving stale content when fresh requests are taking too long to respond
- racing the network against loading from our cache

Google have an excellent project called 'sw-toolbox' [2] which makes implementation straight-forward.

The best part is that if a browser doesn't support Service Workers yet, it'll degrade gracefully. So, whilst visitors won't get the protection, there's no requirement to implement hacky alternatives.

Resource hints

There are a few hints we can provide to browsers to improve both our critical and non-critical path rendering and execution by instructing it of upcoming requests:

- DNS Prefetch
- Preconnect
- Prefetch
- Preload

These are tags analysed by the browsers pre-parser so their appropriate actions can be triggered ASAP without you having to wait for the entire DOM to be compiled.

Note: there is a fifth resource hint, 'prerender', but Google has shown intent to unship it as the overheads outweigh any performance benefits, so I won't be covering it here.

DNS Prefetch

This is the most minor of the three resource hint tags; it simply tells the browser to perform a DNS lookup for a given domain:

```
<link rel="dns-prefetch" href="//example.com" />
```

This means when a request on that domain name is made, there is no wait-time for DNS to be resolved. It'll still require a connection to be initialised though, so this is only useful for times when requests to the domain may not occur on every page load, or when the protocol is unknown (i.e. the request could be over HTTP or HTTPS).

Preconnect

```
<link rel="preconnect" href="https://example.com" />
```

This will make the DNS lookup and open the connection to the server over the given protocol, meaning it's immediately prepared for any requests.

It's perfect for scenarios where requests have dynamic paths or for asynchronous/deferred requests that need to load promptly but aren't on the critical path.

Prefetch

Prefetch is a low-priority hint, browsers will use idle-time to make the request and drop the response into the cache.

```
<link rel="prefetch" href="/next-page.html" />
```

If you know the page that's likely to be the next navigation for the user, you can insert this tag, and it will request the initial HTML before they even click the link.

You don't just have to hard-code this into your HTML; you can dynamically insert these with JavaScript. So you could use hovering an image to trigger prefetching the link it points to, as the user is showing intent they may be interested.

Preload

Preload is, in my opinion, the most valuable resource hint of all.

This instructs the browser to look up the DNS, initialise a connection and start downloading a file.

```
<link rel="preload" href="https://example.com/script.js" as="script" />
```

Note: the `as` attribute needs to be set to a value relevant to the type of resource being requested — otherwise there will be double downloads. Also, fonts require the `crossorigin` attribute to avoid CORS issues — this includes self-hosted font files.

This is useful for A/B testing scripts, for example, which need to execute as immediately as possible, or web fonts buried within CSS, which are valuable to display ASAP to give the best visual experience.

Web fonts

Web fonts are a perfect example of how the web is fast until we bog it down with critical-path resources.

The easiest way to prevent them contributing to a slow experience is just not to use them — stick to a native font-stack if at all possible. Another benefit is that users will experience the website in a font they are accustomed to.

Asynchronicity

Relying on a native font stack may not be acceptable to the client/stakeholder's branding requirements, so if we must use custom web fonts, we need to make sure they are loaded asynchronously.

There is a convenient new CSS property for this called `font-display`, which allows us to control rendering:

```
.custom-font {  
  font-family: Lato, sans-serif;  
  font-display: swap;  
}
```

When set to `swap`, it will render the page with the fallback font and switch it out for the custom one once it's been loaded, avoiding blocking the initial render.

As of writing, `font-display: swap` is only supported in Chrome and Opera, but support is also coming in the next versions of Safari and Firefox.

For other browsers, alternative solutions are available — Zach Leatherman's blog is the go-to destination for this advice [3].

Self-hosting

Many web font solutions, including Google Web Fonts, recommend loading their fonts by linking to an external stylesheet.

By implementing this way, we're not only introducing a single point of failure, but loading requires an additional DNS lookup, another TLS handshake and an extra request too, so it's far more preferable to self-host our font files as a first party resource.

An additional benefit of self-hosting is that, if the hosting stack provides HTTP/2 support, we can push the relevant font files to the user when they request the HTML, ensuring content is rendered with the custom fonts asap and the “flash of unstyled text” (FOUT) is minimised. Just be aware that configuring Push is tricky and your mileage may vary, Jake Archibald has a great article on this [4].

Correct formats

Finally, fonts can come in a variety of formats. WOFF2 is the golden standard at the moment — all major browsers support it and it has a ~30% smaller file size than WOFF.

The only time you’ll need WOFF is for support in IE11 and below.

JavaScript

Serving your JavaScript in the most optimal manner can be an endless task, but just starting with the basics will have tremendous benefits.

JavaScript can be instructed to download asynchronously using the `async` or `defer` attributes on `<script>` tags — simple, right? Not quite.

It's worth noting the difference between these two, as many people see `async` as the successor to `defer`.

- By default, scripts will download, parse and execute synchronously, blocking all browser activity.
- With `async`, scripts won't block while downloading, but they will parse and execute as soon as they can.
- With `defer`, scripts will be downloaded as low priority, but parsing and execution won't occur until full page load.

The temptation, for simplicity, is to bundle a single JavaScript file containing all your dependencies and custom code, then serve it using `async`, but this can place significant CPU load on the browser as it comes to execution. Remember: not everyone is lucky enough to own the latest smartphone.

If our build pipeline supports it, we can export two bundles for site-wide scripts, one of which is for our critical scripts and another which is deprioritised until after page load, serving these as `async` and `defer` respectively, so non-critical scripts do not choke the CPU during initial page load.

Bundling

You can take JavaScript/CSS bundling further by exporting many per-page bundles to reduce the size of assets to be downloaded, parsed and executed on each page.

Tree-shaking

It's not the most straight-forward implementation, and it will vary based on technology stack, but it's possible to run both CSS and JavaScript through a process called 'tree-shaking' — this checks which code is executed and removes the unused parts.

This is particularly useful when loading entire CSS/JS frameworks/libraries when only subsets of the functionality are actually executed.

Image optimisation

Images are not on the critical path for rendering pages, so are often relegated in the minds of web performance experts, but they are still an essential part of the user experience.

Starting with layout images (such as logos, icons, background images) first is a solid bet, they are going to change irregularly, so there is no need to re-optimize. Browsers also typically prioritise them over content imagery when loading.

The best way to optimise content images is via a CDN solution, such as IMGIX [5]. This will automate all the work, but they aren't free and it adds a dependency on a third party service. I'd certainly recommend this approach, but as we're discussing methods that don't require a financial investment, the next best solution is performing optimisation prior to deployment.

For both layout and content imagery, our options are to either integrate a tool like ImageOptim [6] into the build pipeline or download the GUI version and manually process each image.

My team experimented extensively with images for Matalan's website, and these are the techniques we finalised upon:

1. Render at 2x but compress heavily, this will result in a smaller, better quality image when scaled down, even for 1x displays.

The only gotcha here is for sites with very image-heavy pages, you may need to be mindful of memory consumption. As always: make sure you test on real low-powered devices.

2. Serve progressive JPEG as the default format, but serve WebP where a browser supports it. You can use either the Accept header for server-side detection or a <picture> element client-side:

```
<picture>
  <source src='path/to/my.webp' media='image/webp' />
  <img src='path/to/my.jpg' />
</picture>
```

3. Lazy-load any images below the fold. LQIP (low-quality image placeholders) or SQIP (SVG-based LQIP), techniques popularised by Medium.com, will make the experience less jarring:

```
<img src='path/to/preview.jpg' data-src='path/to/real.jpg'
class='js-lazyload' />
<script>
(function() {
    document.querySelectorAll('.js-lazyload').
forEach(function(image)
    {
        image.setAttribute('src', image.getAttribute('data-src'))
    })
})()
</script>
```

Onwards and upwards

As you can see, some simple changes can significantly reduce our dependence on the network, but the work doesn't stop here I'm afraid.

Browser vendors are constantly innovating and improving the toolset available to us, so it's important we use at least a little of the time they save us for keeping up-to-date with the evolving web performance landscape and testing new techniques.

Resources

[1] Harry Roberts

The Fallacies of Distributed Computing (Applied to Front-End Performance)

<https://csswizardry.com/2017/11/the-fallacies-of-distributed-computing-applied-to-front-end-performance/>

[2] sw-toolbox

<https://googlechromelabs.github.io/sw-toolbox/>

[3] Zach Leatherman

<https://www.zachleat.com/web/>

[4] Jake Archibald

HTTP/2 push is tougher than I thought

<https://jakearchibald.com/2017/h2-push-tougher-than-i-thought/>

[5] imgix

<https://www.imgix.com/>

[6] ImageOptim

<https://imageoptim.com/mac>

PERCEIVED PERFORMANCE MATTERS, TOO



The author
Jason Lengstorf

Jason Lengstorf (lengstorf.com) is a developer, designer, author, and friendly bear. His focus is on the efficiency and performance of people, teams, and software. At IBM, he creates processes and systems to Make The Right Thing The Easy Thing™. At all other times, he wanders the earth in search of new and better snacks.

We all know the loading speed of our sites matters. Amazon famously claimed that just one second of added load time would cost them \$1.6 billion in sales each year [1]. As it turns out, perceived performance — how fast a page *feels* — is almost as important as how fast it actually is. (For the science behind all of this, read Denys Mishunov's deep dive [2].)

So even if you work on a site where you have zero access to the server, you can still affect performance — and those effects add up to a better experience for the people using the site, better metrics for the company, and a better-looking portfolio for you.

Performance is more than just delivering assets

Traditionally, performance has been summed up as, "How fast can we get all the resources from the server to the browser?" And while that's still an extremely important part of improving performance, it's no longer the *only* part.

Let's take a look at several strategies we can implement — using only front-end code and tools — to improve the perceived load time of our web apps.

Help the browser load assets faster

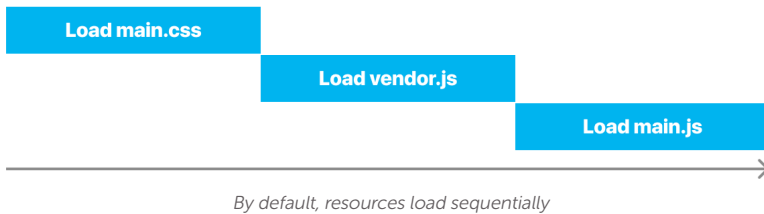
This isn't actually perceived performance — it's a way to get assets downloaded faster. However, it's done entirely on the client side, so we shouldn't skip it.

By default, browsers will typically download resources in sequence, which means that resources are downloaded in the order they appear in the markup. These resources can often prevent the page from rendering, or other resources from loading. This is typically referred to as "blocking".

Consider this markup:

```
<html>
  <head>
    <link rel="stylesheet" href="/css/main.css" />
    <title>My Website</title>
  </head>
  <body>
    <h1>Hello, world!</h1>
    <script src="/js/vendor.js"></script>
    <script src="/js/main.js"></script>
  </body>
</html>
```

On downloading, the browser would get some HTML, pause to load and render the CSS, get more HTML, stop again to load and render the JavaScript, and finally finish loading the HTML.



In modern browsers, we have the ability to start preloading our resources in parallel using `<link rel="preload">` [3], which decreases the amount of time spent waiting for resources to download.

```

<html>
  <head>
    <link rel="preload" as="script" href="/js/vendor.js" />
    <link rel="preload" as="script" href="/js/main.js" />
    <link rel="stylesheet" href="/css/main.css" />
    <title>My (Faster) Website</title>
  </head>
  <body>
    <h1>Hello, world!</h1>
    <script src="/js/vendor.js"></script>
    <script src="/js/main.js"></script>
  </body>
</html>

```

This causes the browser to start downloading the scripts and styles right away, in parallel, which helps speed up the initial render.



For a full breakdown of what `rel="preload"` can do, see this post by Yoav Weiss [4].

Don't let script downloads block the rest of the page

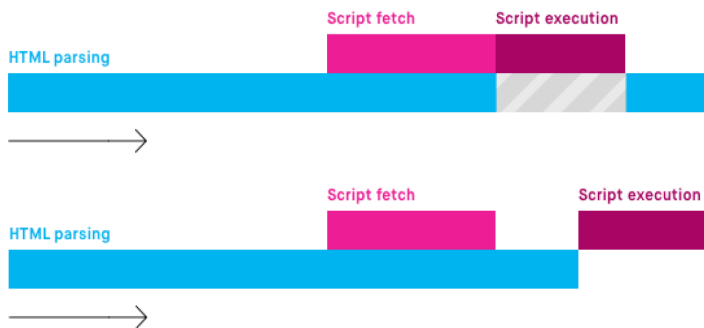
In many cases, we're loading resources that aren't necessarily important for simply getting a page to the browser. For example, if the page has JavaScript to display a modal window after a user clicks a button, that can be downloaded later.

To accomplish this, we can take advantage of the `async` and `defer` attributes [5] on our scripts:

```
<script src="/js/main.js" async></script>
```

Simply by adding this attribute, we tell the browser that it shouldn't wait for the script to download before continuing to render the page. Both `async` and `defer` accomplish this — where they're different is in how they handle the script after it's loaded:

- `async` will execute the script as soon as it's done loading, blocking the HTML parsing until it's finished, which is ideal for critical resources (for example, if you're working on a single-page JavaScript app)
- `defer` will wait to execute the script until the document has finished parsing, which is great for scripts that aren't critical for rendering the page: analytics, scripts that handle specific user actions, etc.



Example of `async` (top) vs `defer` (bottom). Image credit: Ire Aderinokun

For more information, check out Ire Aderinokun's detailed look at the differences between `async` and `defer` [6].

Load less important resources later

Many other resources, such as images, can usually be loaded later. This is often referred to as “lazy loading”, and it’s a powerful strategy for shrinking the initial size of a given page.

The general idea is this: people shouldn’t have to wait for an image to download that’s not even visible to be downloaded before seeing the content



Example of lazy loading from responsive-lazyload.js

There are many ways to approach this, but it generally follows a structure like this:

1. Instead of loading image files, a placeholder is rendered (a blank GIF, for example) in the markup.
2. The path to the image is added as a data attribute.
3. When the image enters the viewport, a script starts downloading the image.
4. Once the image is downloaded, the placeholder is replaced with the actual image.

To quickly add lazy loading for images in your web app, check out [responsive-lazyload.js](#) [7].

Use the skeleton loading pattern if it makes sense

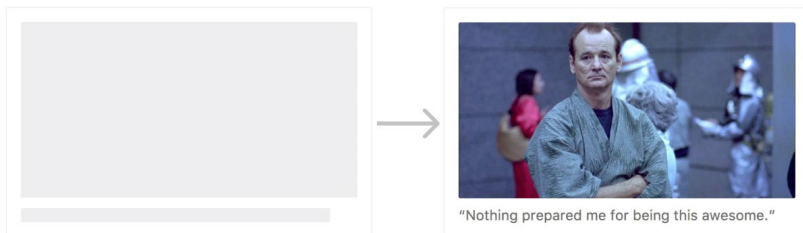
In many apps, the slowest part of the load is the data. Requests to APIs are often sent asynchronously — which is a good thing, because it helps get the rest of the resources to the browser faster. However, this means that often times the markup, scripts, and styles are all loaded, but there's no data to display.

Developers often deal with this by showing loading spinners.



Image credit: <https://loading.io/>

Unfortunately, in some cases a spinner works against the perceived performance of the page. For that reason, many companies — including Facebook, LinkedIn, IBM, Pinterest, and many more — use *skeleton screens*, or mockup-like representations of the interface, to improve the perceived performance of their apps.



Example of a skeleton loading pattern while loading (left) and loaded (right)

Part of what makes this effective is that people are able to start visually exploring the page before data arrives. Skeleton screens let us see roughly where things will be displayed, so we know where

to focus our attention when content arrives.

This requires extra design to implement, but when done well it can have a huge impact on the perceived loading speed of your apps. For a solid walkthrough of creating skeleton components, check out this tutorial [8].

Send fewer requests to the server

One of the best ways to improve load time is to... request less stuff.

There are many ways to accomplish this — many of which are best addressed during design — so let's only look at ways that don't affect the layout of our pages.

Inline small, critical resources

In some cases, swapping out a small CSS include for an inline `<style>` tag can make things faster by removing an HTTP round trip.

However, it's important to weigh the tradeoffs of this approach, because inline resources can't be cached the same way as external files — if the inline resources are large, it can hurt performance.

Google's Lighthouse audit tool recommends inlining critical resources to avoid blocking the render [9], and using `async` or `defer` for all non-critical resources.

Use a Service Worker to make static resources available offline

The addition of Service Worker [10] to most modern browsers [11] is a huge opportunity to boost performance. By adding a Service

Worker, we're able to store static resources offline, which means the browser won't even make a request for cached resources.

This can have huge performance impact, because a many apps are now able to deliver their static resources *only once*, and then make subsequent page loads near-instant, with zero data transfer required to render the next page. (Coupled with the skeleton loading pattern, even apps that need to load data from a server on every load *can feel instant*.)

And thanks to tools like sw-precache [12], setting up a Service Worker is fairly painless in many build pipelines (e.g. Webpack, Gulp).

Explore the benefits of adding GraphQL to your stack

Unlike the other recommendations in this section, GraphQL [13] requires changes to the server, so be sure to weigh the tradeoffs of implementing a new data layer against the benefits of easier data access.

In many applications that rely on REST APIs, it can be tricky to set up and chain together all the asynchronous calls required to load data. And each of those calls is a separate HTTP request. That complexity makes it hard to keep performance at the forefront, because *just making it work at all* is hard enough.

GraphQL is a way to access data that removes all that complexity from the front-end by allowing complex queries to be sent asynchronously in a single HTTP request, using a straightforward approach that makes good performance practices — such as skeleton loading patterns — extremely simple to implement:

```
export default ({ loading, data }) => (
```

```
<div className={`block ${loading ? 'block--loading' : ''}`}>
  {data && <p>{data.content}</p>}
</div>
);
```

For an example of how a GraphQL request can be used with the skeleton component pattern in React, check out this example on CodePen [14].

GraphQL is not magic, though, so the complexity this removes from the client-side is transferred to the server side. I tend to prefer this, because it makes a clearer separation between presentation logic and business logic, but whether or not the benefits are worth the effort will vary from project to project.

Remember: don't skip actual performance

Addressing client-side performance is critically important, but remember that perceived performance can't fix underlying performance issues — it merely masks them. So if you're working on an app with slow server response times, the techniques in this article should be used *in addition* to efforts to speed up actual performance, not *in place* of them.

Server performance is beyond the scope of this article, so we won't cover it here. For some ideas to get started, take a look at HTTP/2 [15], using CDNs [16], and caching server responses [17].

Resources

[1] Kit Eaton, FastCompany

How One Second Could Cost Amazon \$1.6 Billion In Sales

<https://www.fastcompany.com/1825005/how-one-second-could-cost-amazon-16-billion-sales>

[2] Denys Mishunov, Smashing Magazine

Why Perceived Performance Matters, Part 1: The Perception Of Time

<https://www.smashingmagazine.com/2015/09/why-performance-matters-the-perception-of-time/>

[3] W3C

Preload

<https://w3c.github.io/preload/>

[4] Yoav Weiss, Smashing Magazine

Preload: What Is It Good For?

<https://www.smashingmagazine.com/2016/02/preload-what-is-it-good-for/>

[5] Mozilla

<script>: The Script element

<https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script>

[6] Ire Aderinokun

Asynchronous vs Deferred JavaScript

<https://bitsofco.de/async-vs-defer/>

[7] Jason Lengstorf

Responsive Lazyload Examples w>Loading Animation

<https://code.lengstorf.com/responsive-lazyload.js/loading-animation.html>

[8] Max Block, CSS Tricks

Building Skeleton Screens with CSS Custom Properties

<https://css-tricks.com/building-skeleton-screens-css-custom-properties/>

[9] Lighthouse, Google

Render-Blocking Resources

<https://developers.google.com/web/tools/lighthouse/audits/blocking-resources>

[10] Mozilla

Service Worker API

https://developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API

[11] Can I Use

Service Workers

<https://caniuse.com/#feat=serviceworkers>

[12] sw-precache

<https://github.com/GoogleChromeLabs/sw-precache>

[13] GraphQL

<http://graphql.org/>

[14] Jason Lengstorf, CodePen

Skeleton Loading Pattern with CSS

<https://codepen.io/jlengstorf/pen/NabRVb>

[15] Ilya Grigorik and Surma, Google Web Fundamentals

Introduction to HTTP/2

<https://developers.google.com/web/fundamentals/performance/http2/>

[16] Mozilla

CDN

<https://developer.mozilla.org/en-US/docs/Glossary/CDN>

[17] Akos Kemives, RisingStack

Redis + Node.js: Introduction to Caching

<https://community.risingstack.com/redis-node-js-introduction-to-caching/>

THE NEXT FOUR BILLION: HOW TO MAKE SITES PERFORMANT ON MOBILE DEVICES



The author
Jem Young

Jem Young (jemyoung.com) is a (very) tall engineer at Netflix who loves dogs, reading, and clean code. He really enjoys working across the stack but his true passion lies in JavaScript and building a clean user experience. He believes that empathy is the key to building an effective UI and when he's not out riding his bike, you can find him encouraging other engineers to write more tests.

By the end of 2017, around 40% of the world's population will have access to the internet. Thanks to the proliferation of cheap smartphones and expanding global coverage, the vast majority of people will be on mobile devices so it's not a surprise that mobile usage continues to grow each year. Now more than ever, it's crucial to deliver a lightning-fast user experience, especially in emerging markets such as India or southeast Asia where a phone can often end up being the sole device for communication, entertainment, and information. As engineers, we need to treat mobile devices as first class citizens and optimise for the next four billion people, who will be connecting to the internet using a mobile device.

Limitations

Before we can talk about performance, we need to define what exactly constitutes a mobile device. Though mobile computing has exploded in the past two decades and the precise definitions are a bit murky, we can broadly categorise something as "mobile" if it's a network-capable device, has a screen, is portable, and runs on a battery. From this definition we can extrapolate what our goals should be when we talk about mobile performance:

Performance goals for mobile devices

- Use as little data as possible
- Use as little CPU and battery as possible
- As fast as possible

When thinking about mobile users, we have to understand how mobile computing compares to desktop computing. Aside from screen size, the two primary differences are connection speed and

processing power (how long it takes to download the assets and how long it takes to execute the code). While desktops and laptops tend to connect to the internet via WiFi or Ethernet, a mobile device is typically on a cellular network where download speeds are constrained and often limited by the budget of the consumer as they have to pay per byte of data transferred. Another constraint to consider is the computational ability of the device; that is, the availability of processing power in terms of CPU and battery usage. Now that we understand the constraints of working with mobile devices and with our performance goals in mind, we can come up with two simple rules for mobile performance:

Rule #1: Send as little data as possible to the client

Rule #2: Don't waste processing resources

Now, let's keep these two rules in mind as we cover some practical performance optimisations for mobile.

Serving the right asset

When considering connection speed it's important to remember that most of the world is on slow and often unreliable cellular connections. As developers, we often think nothing of adding a useful node module or two as the need arises to help us in the relentless bid for inertia that is coding "in the zone." Don't forget that the true cost of such actions are felt by the end user, especially those on mobile connections where an extra 500KB can sometimes add three to four seconds to a page's loading time. By taking a holistic view of our web applications and understanding how the HTML, CSS, images, and JavaScript fit together, we eliminate the unnecessary parts and create a better experience for mobile users. The best place to start is by examining the image assets that make up your websites' payload.

Take a look at these three pictures:



Three images at different resolutions. Photos by burak kostak from Pexels

A 5184 × 3456

B 1920x1080

C 640x426

On a typical mobile device, all three images look fairly identical. On a high-resolution screen, say a 15" MacBook Pro retina display, images B and C appear fuzzy as they have a lower resolution than the native 2880x1800 of the MacBook. Unfortunately, many engineers and designers use high-resolution images when designing applications and forgetting that while a high-resolution image looks great on a large screen, it's overkill on most mobile devices. This breaks Rule #1 — send as little data as possible to the client. Fortunately (and I don't say this very often), HTML to the rescue! Using a standard `` tag, we can define rules to help the browser know which image asset to download and display.

```

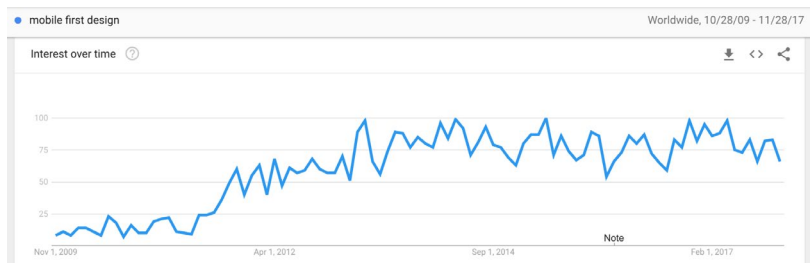
```

Using the `srcset` attribute, we can now tell the browser which image (A) is best suited for a mobile device, and this nets us a decrease of about 600% in file size when compared with image B.

Using the `` tag with `srcset`, it's easy to ensure that the best fitting image asset is sent to the client. What about images loaded from CSS? Next, we'll take a look at media queries.

"Mobile first" design

The term mobile first design has been a part of the UI engineering lexicon long enough for the message to have (mostly) sunk in: we have to always consider the mobile user experience.



Google trend for the popularity of the term "mobile first design"

When most UI engineers hear the term, they immediately think of media queries which are essentially a way to set CSS rules based on the user's device parameters. A standard CSS rule for setting a background image to the body of a page might look something like this:

```
body {  
  background: url('images/bg/hi-res.jpg');  
  background-size: cover;  
  @media only screen and ( max-width: 600px ) {  
    background: url('images/bg/lo-res.jpg');  
  }  
}
```

Modern web browsers will intelligently parse this CSS, and only the image for which the rules apply will be downloaded. In this case, a browser on a mobile device will only download the lo-res.jpg image. When thinking about mobile first design, it's important to not only consider the layout of your application but also the layout of your code. The browser can give us easy performance wins if we structure our CSS correctly.

Sending the correct image asset to the user is one of the fastest and easiest ways to improve performance on mobile devices.

Browser hinting

Though a user's connection and CPU time are finite, we can use a series of techniques collectively known as "resource hinting" to help tell the browser about what assets the user is going to need next. While the user is reading or otherwise interacting with your page, the browser can fetch static assets in the background, so that when the user navigates to the next page in your app, the assets needed to render the page are already in the browser's cache, which increases the perceived performance of the page. Resource hinting is an especially effective technique for mobile users when the connection is limited, and making excessive network requests cannot only slow the current page down, it can make the page appear as if it is still loading, which is a poor user experience. While there are a few types of resource hints [1] that we can use,

we're going to focus on two of the most commonly used types: prefetching and prerendering.

Prefetching

Prefetching a resource means telling the browser that the user will need the asset in the near future. If the browser has idle time, it will make a network request in the background and add that resource into the cache. To utilise prefetching, we add an HTML `<link>` tag with `rel="prefetch"` into the page markup.

Example: prefetching a CSS file

```
<link rel="prefetch" href="/static/index.css" />
```

Now, when you inspect the network, you'll see a request for `index.css`. We can identify prefetch requests by inspecting the request headers and looking for a `purpose: prefetch` or `X-moz: prefetch` header. It's important to note that when prefetching a resource, the browser does not do any evaluation. For example when the user taps the link to move onto the next page, there will still be a brief delay, as the browser has to evaluate the HTML/JavaScript/CSS before rendering the page. This is where prerendering comes in.

Prerendering

Prerendering, as the name implies, renders an HTML page ahead of time, before the user navigates to it. In the mobile computing world where CPU time and network speed are limited, the ability to fetch and render a web page in the background can be a valuable tool. If the browser has idle time, it will make a network request in the background for an HTML page and related assets and render it in a hidden tab. To utilise prerendering, we add an HTML `<link>` tag with `rel="prerender"` into the page markup.

Example: prerendering the next page

```
<link rel="prerender" href="/nextPage.html" />
```

Unlike prefetching, browsers don't send any special headers when making a prerender request. To identify such requests, we have to utilise the Page Visibility API [2] to register when a page was prerendered. Here's an example of such code:

```
document.addEventListener("visibilitychange", () => {  
  if (document.visibilityState === 'prerender') {  
    // Log that the page prerendered  
  } else if (document.visibilityState === 'visible') {  
    // Start animations, videos, logging, etc...  
  }  
}, false);
```

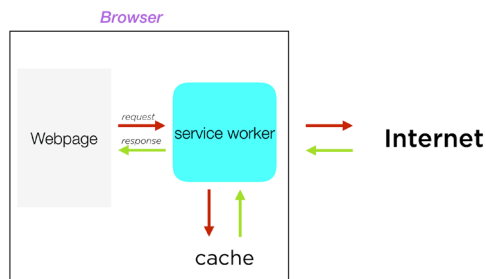
While prefetching and prerendering are amazing tools, there are caveats that affect our two main concerns when it comes to mobile computing: network requests and resource consumption. With both prefetching and prerendering, we need to be very certain that the user is going to actually use the resources that we've fetched ahead of time. Otherwise, we're just wasting the user's network connection which is especially important to people on cellular connections, where the data plans are often pay-per-byte. Even accounting for the network cost, prerendering may seem like a silver bullet when it comes to mobile performance: the user taps on a link and the page is available instantaneously but we have to consider our second rule — don't waste processing resources. Constantly prerendering the next page means that we're always rendering an extra page on every request, and that essentially means that we're using twice the CPU, which can have an adverse effect on the user's battery life.

Only utilise prefetch/prerender resources if you have a high degree

of confidence that those resources will be used in the near future.

Progressive Web Apps

Progressive Web Apps (PWA) are one of the most exciting things to happen to the web since the introduction of AJAX. As engineers, we can now create web apps that look and feel like native applications, work offline, and progressively enhance based on the capabilities of the user's browser. One of the most valuable features for mobile users is the ability to cache content, so that your application not only loads quickly, it has the ability to work offline thanks to the persistent Cache API. At the backbone of all PWAs is a Service Worker, which has all the capabilities of a standard Web Worker with the added benefit of persisting outside the tab context and having the ability to intercept and cache network requests.



A Service Worker can intercept and cache network requests to allow your app to load faster or even work offline

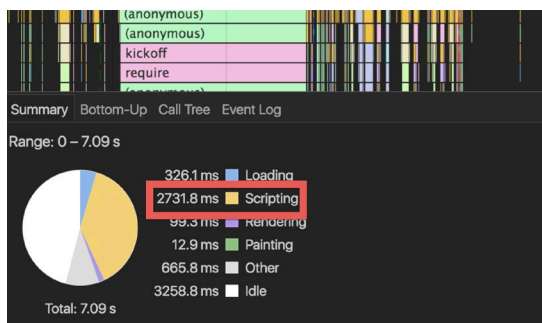
Using a Service Worker to cache requests means that we can now build an application that works offline or, in the very least, loads much faster thanks to resource caching. A site that has many repeat visitors and a high degree of interactivity or static resources

is a good candidate for using a PWA. Mobile users will especially benefit as caching means fewer network requests and thus a faster experience overall. However, using a PWA has several costs including cache management, worker persistence, and adding additional JavaScript code to your application's payload. Thus, if your site does not have many repeat visitors and does not have a lot of shared code between pages, a PWA may not be worth the cost. A better decision for sites like this would be to actually simplify and convert it into a static site (i.e. little to no JavaScript or XHR).

Static site case study: Netflix

The Netflix homepage is one of the top 50 most visited sites on the internet, so naturally, they run a fair number of performance-based A/B tests, constantly testing various optimisations to improve the experience for millions of visitors each day. In one of those tests a colleague of mine, Tony Edwards [3], removed most of the client side JavaScript, including the UI library and rewrote all the necessary code (event handling, logging, etc) in vanilla JavaScript.

Tony studied all of the UI code and realised that other than for server-side rendering, there wasn't actually much use in including most of the JavaScript that was being sent to the client (Rule #1). If Netflix could send less code to the client, not only is that a win in terms of network usage, it's a win for processing time, which in turn decreases battery consumption. If you run a performance profile on most modern web pages, you'll notice something curious: most of the processing time is not spent on the loading or the rendering of the page, it's spent just reading and parsing the JavaScript payload.



A breakdown of processing time on a web page

By relying on only server-side rendering to create what is essentially a static page, **Netflix was able** to shrink its payload and decrease network and CPU usage.

Netflix was able to drastically cut its JavaScript payload by not including a client-side UI library, which led to a 50% reduction in Time to Interactive on their homepage.

Aristotle once said, “nature operates in the shortest way possible” and this is exactly how we should build for mobile performance: only send what is absolutely necessary to get the job done. By simply reducing the payload, we can achieve a twofold success for mobile users as we’re using less data and less battery.

Conclusion

Now that we understand how to serve the right image assets, the use cases for progressive web applications, and how simply converting a web page into a static site can improve performance, it's easy to see that making our sites performant on mobile devices is a very achievable goal. The most important thing to remember is to build with empathy. As web engineers we have to imagine what it would be like to only have internet access via a low-end phone and slow network because that's the status quo for most of the world today. We now have the tools and knowledge to build a faster, better internet for not only those on the web today but for the next four billion coming online very soon.

Resources

[1] W3C

Resource Hints

<https://w3c.github.io/resource-hints/>

[2] Mozilla

Page Visibility API

https://developer.mozilla.org/en-US/docs/Web/API/Page_Visibility_API

[3] Tony Edwards, Twitter

<https://twitter.com/tedwards947>

THE CRITICAL PATH: A QUEST TO RENDER PIXELS QUICKLY



The author
Stefan Judis

Stefan Judis (stefanjudis.com) started programming six years ago and quickly fell in love with web performance, new technologies, and accessibility. He worked for several start-ups in Berlin and recently joined Contentful to tell the world how an API-first CMS can make you a bit happier. He is also the curator of the web performance online resource Perf Tooling (perf-tooling.today), organiser of the Web Performance Meetup Berlin, contributes to a variety of open source projects and enjoys sharing nerdy discoveries.

The times in which front-end developers had to explain that mobile matters are finally over. Everybody understands that products that work well on mobile will be more successful.

Especially with a global target audience this is very important. Today, the country with the most internet users is China (751 million people) [1] and 96 percent of the people in China use the “mobile internet” [2]. You cannot serve these users with a broken mobile experience.

But what does that actually mean — “works well on mobile”?

Personally, I don’t really like the differentiation between mobile and desktop environments and prefer to “just build performant sites”. If a site feels fast on a mobile device on a mobile connection, it will feel even faster in a desktop environment.

Mobile taught us an important lesson, though. Connections are flaky — there is no such thing as a stable internet. We have to give our best to get something on the screen as quickly as possible to guarantee a good user experience.

To understand how to achieve a quick rendering, we have to understand how browsers work and what it takes to render pixels on the screen.

The construction of the DOM and CSSOM

Before a browser can display a website on the screen, there is a lot of work involved. It all starts with the initial request for the HTML document. The browser then receives the bytes, converts these to characters, transforms them to tokens defined in the HTML spec

[3] and continues to convert them to objects representing the actual HTML elements like paragraphs or images.

Due to the structure of HTML, the work is not done at this point. The created objects are heavily nested, which means one object can include other objects, which then can include more objects. Think of a paragraph that includes images and spans. The final step is to create a tree representation of all these objects to keep the information of how they all relate to each other. This tree is called the **DOM (document object model)** and is used for all further browser rendering operations.

Unfortunately, the DOM tree doesn't include any information on what the parsed HTML should look like and how the elements should be laid out. This information is included in external stylesheets, inline style elements or style attributes. The operations that are needed to bring these styles into a usable format are similar to the ones needed to construct the DOM.

The browser scans the discovered CSS and also transforms it into a tree structure. The result is called **CSSOM (CSS object model)**.

Structural and visual information is now available in form of the DOM and CSSOM, but these two are completely independent and not connected to each other. To get the first pixels onto the screen, there is one final step to perform.

The required combination – the render tree

To figure out what should be rendered, the browser walks through the DOM tree and evaluates what elements should be visible by looking up the matching tree nodes defined in the CSSOM. Elements like the head or elements that have a matching display: none; CSS declaration won't be included in the render tree. By

combining the information of the DOM and CSSOM the render tree includes everything that is needed to render and lay out every visible element.

This whole process is called **the critical rendering path** or in short the critical path. As you have seen, there are several steps that are included in **the critical path**. The time it takes significantly depends on how the website is structured, but also the device capabilities and network conditions of the user.

How can we shorten the time duration of the critical path and create a good user experience?

Minimise the render blocking

When analysing the critical path for a website the first thing I recommend is to check what is included in the head element of a site. There are several ways to increase the time to first render and very often the most important blockers can be found in the head.

The critical render tree setup very much depends on the fast creation of the DOM and the CSSOM. You'll want to avoid anything that could interfere with the work of the DOM or CSSOM parser.

So what should you watch out for?

Scripts – the synchronous delayers

JavaScript interrupts the browser in the creation of the render tree, which is a common problem. There are several ways to include and execute JavaScript code in your documents. The first one is a simple inline script element.

```
<head>
  <script>
    console.log('Hello world!');
  </script>
</head>
```

You can place this wherever you want and the browser will execute the script. JavaScript is a dynamic language and has access to the DOM and the properties that are defined in the CSSOM. As this inline script can change the DOM structure, it interrupts the creation of the DOM and browsers execute the script when it's discovered. As you can imagine, this slows down the creation of the render tree. The execution of the example might only take a few milliseconds but still... it stops and slows down the parsing of the HTML.

Let's have a look at another example:

```
<head>
  <script src="../../some-javascript.js"></script>
</head>
```

script elements can also request complete JavaScript files. This means the browser has to fetch an additional resource. The behaviour remains the same, though. The browser interrupts the HTML parsing and requests the file to execute it. The delay now increases from a few milliseconds to whatever time it takes to fetch the file. Think of your mobile connection now, under certain circumstances this can take several seconds and is something to avoid at all cost.

```
<script src="./some-javascript.js"></script>
```

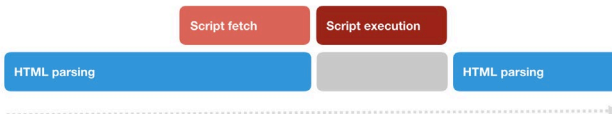


*The blocking of the HTML parser with a synchronous script
(blocked during download and execution)*

This delay is the reason why it was best practice to put script elements at the end of the file in order to not interrupt the DOM creation at the beginning, but rather at the end. However, this is not best practice anymore and there are better ways to do it.

script elements also support attributes called defer and async. These attributes change the script execution behaviour of the browser to not interrupt the DOM creation.

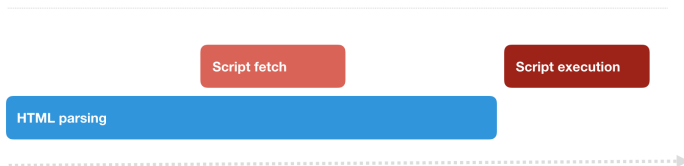
```
<script src="./some-javascript.js" async></script>
```



The blocking of the HTML parser with an asynchronous script (blocked during execution)

Scripts with an async attribute are fetched asynchronously so that the browser does not stop to parse the HTML while the file is downloaded. There is one catch, though. Whenever the script download finishes, the parser will be stopped and the script executed. This means that asynchronously loaded scripts usually don't delay the first render but can interrupt the rendering in case the HTML is not completely parsed yet.

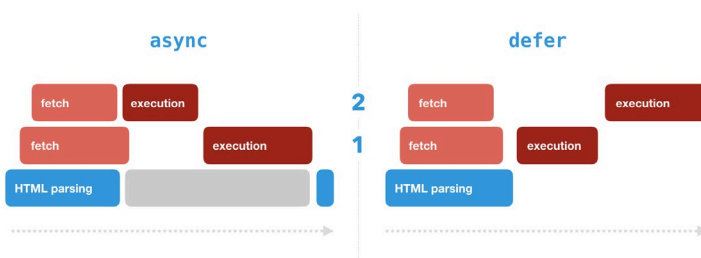
```
<script src="./some-javascript.js" defer></script>
```



The non-blocking of the HTML parser with a deferred script

Deferred scripts also don't stop the HTML parser when they're discovered but execute only when the HTML parsing is done. This means that the execution of deferred scripts can happen later than asynchronous scripts but these scripts don't interfere with the DOM creation, which is a good thing.

There is one last big difference between `async` and `defer` that has to be mentioned. Multiple `async` scripts are executed whenever they are downloaded no matter of the order in the HTML document. Deferred scripts, on the other hand, keep the order in the document. This can be extremely handy when dealing with several included scripts that rely on each other.



The comparison of the execution order scripts requested by the `async` and `defer` script elements

But why are synchronous scripts at the end of the file an anti-pattern today?

The reason is that scripts at the end of the document will be discovered later. Imagine having to deal with a massive HTML file. The browser concentrates on all the markup first and then discovers that there is a script at the end. This delays the request creation, whereas a deferred script can be requested and executed as soon as the DOM is ready. If you want to read more on that topic, Steve Souders wrote a great article about it [4].

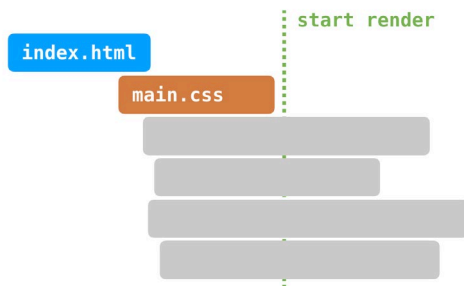
Remember: JavaScript can block the HTML parsing.

Stylesheets – the big blockers

To build the CSSOM the browser has to request all external stylesheets that are included in the document. The rendering can't start before the CSS is being downloaded. That's why external stylesheets are treated as "render-blocking" resources.

This can be problematic. When the browser discovers an external stylesheet in a link element, it has to be requested (if it's not sitting in some cache), parsed and transformed to the CSSOM first.

These three tasks heavily rely on network conditions and device capability. The time needed to request a CSS file on a slow 3G connection is affecting the critical path in a similar way as a huge CSS file that has to be processed by a low-end device. That's why a critical look at the loaded CSS is always valuable.



Start render time with a render-block external stylesheet

Bringing in complete libraries like Bootstrap or Foundation with a simple link element to have a few lines of CSS for a few button styles come with a cost. Maintainable and well-structured CSS does not only make developers happier — cluttered CSS with tons of overwrites results in more render-blocking bytes on the wire, too.

Remember: the amount of CSS you ship has a direct impact on the user experience.

Get everything down as quickly as possible

So, if you rely on the network, it brings in factors you can't control and can drastically slow down the critical path. What if you could get around the blocking behaviour of external CSS to speed up the first render?

Inlining of critical CSS

You can save requests and avoid network dependencies by inlining resources in the HTML document via style and script elements

or by using encoded images. This might sound like a good idea but should be considered very carefully. Inlined resources like a complete stylesheet can't be cached separately and will result in more bytes that need to be downloaded at every visit.

What if you could only inline the styles that are needed for the first paint and load the rest asynchronously to speed up the first render?

To evaluate the required styles you can either do it by hand or use projects like Addy Osmani's Critical [5]. Critical evaluates the styles to display the content that is visible "above the fold". "Above the fold" styles are the styles that define how the initial visible area of a site looks when a user opens it in their browser. In responsive web design, the evaluation of these can be tricky because the page might look different depending on the device but with a bit of trying and tweaking it can be absolutely worth it.

Critical also offers a ready-to-use workflow to extract these critical styles, inline these into the document and load the remaining CSS asynchronously into your project.

Loading CSS asynchronously is possible with the smart use of a link element and a `rel="preload"` attribute. The `preload` value lets you specify resources that will be needed for a document. The browser then starts downloading the stylesheet and with the help of an `onload` handler the `rel` attribute can be changed to `stylesheet`, which means that the styles are applied when the resource is available. This way you can load CSS in a way that's not render-blocking.

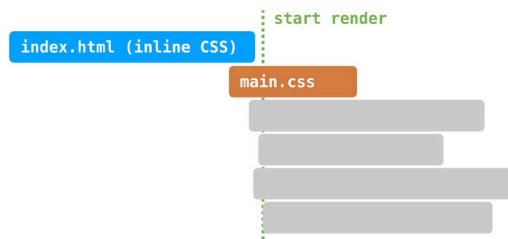
```
<head>
  <style> // critical styling </style>
  <link rel="preload" href="path/to/mystylesheet.css" as="style"
```



```
onload="this.rel='stylesheet'">
</head>
```

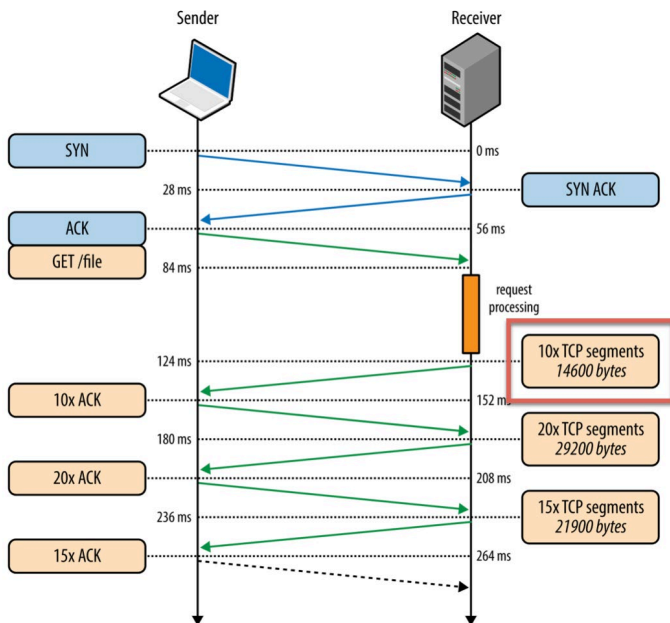
The Filament Group is maintaining the loadCSS project [6], which is the go-to resource to implement the loading of asynchronous CSS.

This technique can help to drastically decrease the time that passes before the first render because there is no render-blocking stylesheet included in the document anymore.



Start render time with inlined critical CSS and an asynchronously loaded stylesheet

The optimal result can be achieved when the inlined CSS has a size of not more than 14 kilobytes. Why 14kb? Let's have a detailed look at the functionality of HTTP.



The anatomy of an HTTP request including TCP setup

An HTTP request consists of the TCP handshake (SYN, SYN ACK & ACK) that has to be performed before a resource can be downloaded.

The download starts using a principle that is called "TCP slow start". The idea is that the server starts sending a relatively low number of bytes first (14,600 bytes in the first round) and increases the amount of transmitted data with every run until packets get

lost. This way the network's maximum carrying capacity can be evaluated.

By fitting all the required resources into the first round of received bytes the optimal first render time can be achieved because no further round trips are needed. This way the critical path is as short as possible. If you want to read more about the details of HTTP and networks in general, Ilya Grigorik published a whole book that's available for free online [7].

Inlining? Can't we simply use http/2 push?

A lot of people suggest using http/2 push in order to not have to rely on inline styles. This is correct in theory but there are some caveats with http/2 push today.

The optimal push is not there yet

http/2 with all its benefits (for example, hpack header compression and multiplexing) has broad support in the current browser landscape. One of its praised features is the "push" functionality. The idea is that the server can push resources to the browser without it asking for it.

The scenario could be that a browser requests an HTML file and the server responds with the HTML, but also with all the required CSS and JavaScript files at the same time. This sounds great, but the problem is that there is currently no way for the server to know what data is available in the cache of the browser. It simply pushes everything... every time. This results in wasted transferred data.

Yoav Weiss wrote about this topic years ago in his article "Being pushy" [8] and recommends using http/2 push only for resources that would usually be inlined — for example, the critical CSS.

If there is some server-side processing involved, which delays the first bytes of the HTML document (most importantly the head), then http/2 push can bring some benefits over the inlining of resources. Otherwise, the difference between pushed and inlined critical CSS is very small. In the end, it's your call if the relatively complex implementation of http/2 push is worth it.

rel="preload" to the rescue

The fact that the server using http/2 push doesn't know the content of browser caches (yes, there are many [9]) is a big drawback, but there is another technique that can help to get resources down the wire faster.

You remember the rel="preload" hack to load stylesheets asynchronously? Let's have a look at what preload should actually be used for. It can be used as an element included in the head...

```
<link rel="preload" href="https://fancy-fonts.io/some-font.woff2"
      as="font">
```

or it can even be set as an HTTP header...

Link: <https://fancy-fonts.io/some-font.woff2>; rel=preload; as=font;

Each implementation gives the browser information on what resources are needed in the document and should be requested early. Let's take web fonts. Usually the browser has to download the CSS and create the CSSOM and render tree to then figure out if web fonts are needed for the document and to download the file. Using rel="preload" it can start the download right away knowing that it needs them at some point.

In comparison to http/2 push, preload will always be slower because the browser has to react to the HTML response of the server, whereas push would send the data right away. What's really powerful about preload, though, is that it takes the browser cache into consideration.

This means that preload is only a tiny bit slower than http/2 push but easier to implement and debug, and it doesn't waste bandwidth with already cached resources.

http/2 cache digests – the glory future

A new specification called "cache digests" [10] is in development to solve the "unknown cache" problem of http/2 push. It defines a way, in which the browser can inform the server about its cache internals and contents. This way the wasted bytes of http/2 push can be saved and resources can really be served before the browser asks for it — which will result in a huge performance boost.

Optimising the critical path

Optimising the critical path is all about getting rid of render-blocking resources and the creation of well-structured documents:

- Focus on the critical resources
- Get required assets down first and get them down quickly
- Avoid additional networks dependencies and ship as little initial data as possible

These steps might seem like a lot of work but a fast first render time will make a big difference to your users. Every additional second of waiting leads to frustration and increases the chance that your users will simply leave before your site loads. Today, expecting that your users will wait for your site to load is simply ignorant, and will make them annoyed and lead them to your competitors. People have better things to do. You know how it is: "ain't nobody got time for that"!

Further reading

- Google published a complete series including tips and tricks on how to optimise the critical path. [11]
- Patrick Hamann constantly educates, teaches and speaks about critical path optimisations. His talks are highly recommended. [12]
- Ben Schwarz wrote an excellent article on request prioritisation, critical requests and ways to download them as quickly as possible. [13]

Resources

[1] Internet World Stats

Top 20 Countries in Internet Users vs. All the World – June 30, 2017
<http://www.internetworldstats.com/top20.htm>

[2] China Internet Watch

China Internet Statistics 2017
<https://www.chinainternetwatch.com/whitepaper/china-internet-statistics/>

[3] W3C

HTML 5.2
<https://www.w3.org/TR/html5/>

[4] Steve Souders, Performance Calendar

Prefer DEFER over ASYNC
<https://calendar.perfplanet.com/2016/prefer-defer-over-async/>

[5] Critical

<https://github.com/addyosmani/critical>

[6] loadCSS

<https://github.com/filamentgroup/loadCSS>

[7] Ilya Grigorik

High Performance Browser Networking
<https://hpbnp.co/>

[8] Yoav Weiss

Being Pushy
https://blog.yoav.ws/being_pushy/

[9] Jake Archibald

HTTP/2 push is tougher than I thought
<https://jakearchibald.com/2017/h2-push-tougher-than-i-thought/>

[10] HTTP Working Group, Internet Engineering Task Force

Cache Digests for HTTP/2
<https://tools.ietf.org/html/draft-ietf-httpbis-cache-digest-00>

[11] Ilya Grigorik, Google Web Fundamentals

Critical Rendering Path

<https://developers.google.com/web/fundamentals/performance/critical-rendering-path/>

[12] Patrick Hamann, CSSconf EU 2017

CSS and the first meaningful paint

<https://www.youtube.com/watch?v=4pQ2byAoIX0>

[13] Ben Schwarz, Medium

The Critical Request

<https://medium.com/@benschwarz/the-critical-request-90bb47da5769>

OPTIMISE PRIME: HOW TO OPTIMISE IMAGES FOR PERFORMANCE



The author
Henri Helvetica

Henri (twitter.com/HenriHelvetica) is a freelance developer who has turned his interests to a potpourri of performance engineering with pinches of user experience. When not reading the deluge of daily research docs and case studies, or indiscriminately auditing sites in Dev Tools, Henri can be found contributing back to the community, co-programming meetups including the Toronto Web Performance Group or volunteering his time at various bootcamps. Otherwise, he's training and focusing on running the fastest 5k possible.

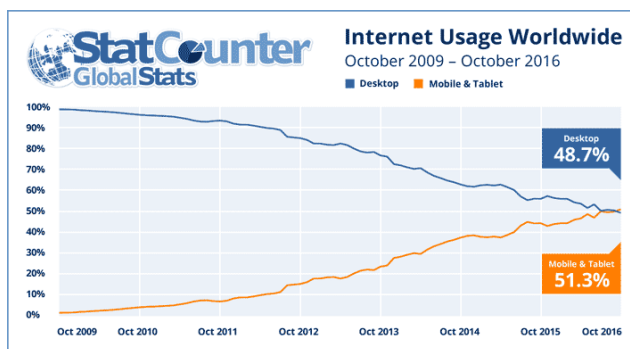
When Steve Jobs unveiled the iPhone in 2007, he said the following during the demonstration:

"I'm going to load in the New York Times. It's kind of a slow site 'cause it's got a lot of images."

This was a comment that largely escaped attendees during the historic unveiling. However, a decade later, we are arguably having the same difficulties.

Over the last few years we have also seen an explosion of the use and power of digital photography, thanks to the ubiquitous and omnipresent smartphone. Users not only shoot but also share these images online. Pictures have become literal currency. Using this currency has proven to be a challenge.

Recent data published indicated that for the first time in history, the mobile phone surpassed the desktop as the primary device to access the internet.



*Worldwide internet usage chart: desktop vs mobile and tablet from
October 2009 until November 2016*

It's ironic then that the very devices creating much of the photographic content are also the ones struggling to process these digital assets adequately. The average smartphone is a \$150 to \$200 mid to lower tier Android device [1].

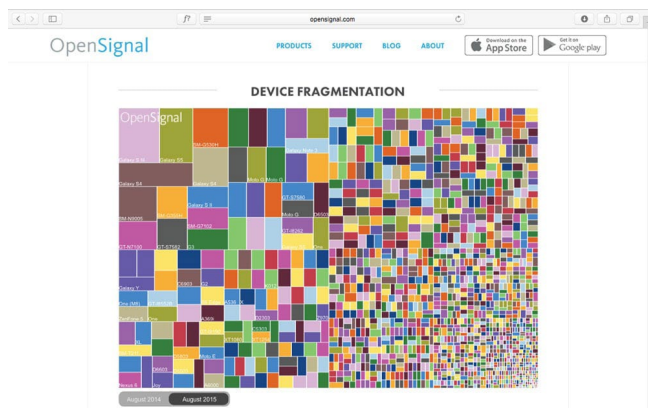
The front-end is the most hostile development platform in the world [2], and images are part of that maelstrom. When speaking of performance, the case for the proper management of images and their optimisation can be made in many ways.

The reason for image optimisations

We must **optimise for the best user experience**, and much of that is based around speed. Facebook recently decided to show people more stories that will load quickly on mobile and fewer stories that might take longer to load [3].

We must **optimise for the networks** we use. A classic study told us that 53% of visits are abandoned if a mobile site takes more than three seconds to load [4], and networks do not handle large images well.

As we are working with mobile, we must **optimise for screens**. Responsive design and device fragmentation have made image management a touch more challenging.



Android phone device screen fragmentation

Improperly sized images are a large part of the challenges online. It leads to many hurdles, including the following: image decoding and memory management.

The image decoding can be very taxing and work is being done to alleviate the stress. [5]

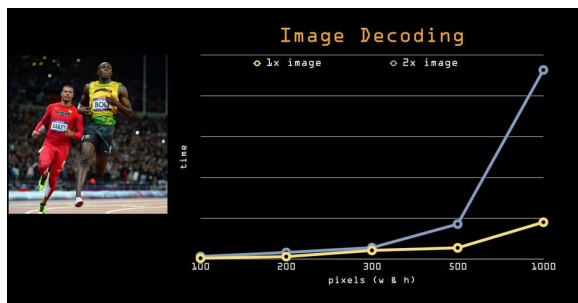
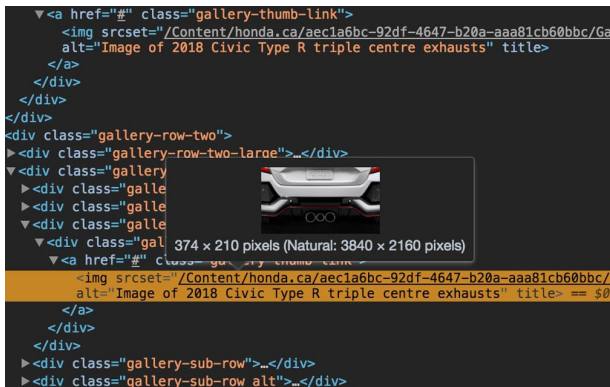


Image decoding chart indicating size vs decoding time

The greater the amount of pixels, the more time consuming the decoding process. As such, a poorly sized image is taxing.

Below is an example of a car manufacturer's site and a desktop-sized asset loading on mobile, forcing for more decoding than necessary.



A desktop size image asset shown loading in mobile

Also keep in mind the memory. Depending on the colour space, the memory footprint of the image can be substantial. To quote Ilya Grigorik of Google: "Out-of-memory errors are very common in low end devices, very very very common and the number one culprit is images." [6]

All of these have real world implications. Images are still an achilles heel at retail [7], and the data costs of images are also a hurdle for users.

At writing, the average page weight is 3,464KB with 53% of it comprised of image assets. As the largest source of data, images will also be the largest source of data savings.

Image formats

Understanding images [8] and the available formats is part of the optimisation process when making performance decisions. So let's look at the main formats used today.

GIF

The Graphical Interchange Format, the one that is best known for being pronounced with a soft 'G', has also been around the longest. Introduced in 1987, it's a lossless format capable of 256 colours and transparency, making this one good for small logos and the like.

Conceived in the early days of computing, its current reputation for bloat makes it the least favourite and viable of the usable formats. More recent advancements in computing have allowed for some lossy optimisations for the old-school hold overs. Only in small and rare cases however might this be the smallest of options to employ. Even the popular animated GIFs found online are essentially now MP4s. That said, the GIF has been called a useless format.

SVG

The only one of the non-raster image formats to be actively used online, the Scalable Vector Graphic, which dates back to 1999, is only now seeing a spike in popularity. This vector format by design compresses well as it's not storing any pixel data, but mathematical detail of colours, lines, curves etc... textual information. As such, SVGs can be compressed and optimised using gzipping [9] or even brotli [10].

Best for logos, icons, simple graphics and the like, the key is to reduce the SVG's complexity as much as possible. That means reducing the precision, collapsing groups, rounding numbers (reducing the number of decimal places) and other things like removing the metadata added by the authoring software. As the complexity is simplified, the taxing of the GPU lightens, which is especially useful for older devices.

A great tool to further optimize the SVG is the SVG Optimizer or the SVGO [11]. This node tool also comes as a web app, SVGOMG [12], grunt and gulp tasks, with additional plugins for popular apps like Illustrator and Sketch.

PNG

The Portable Network Graphic is the second most popular image format. It came about during a patent spat with the GIF format. As such, the PNG was born with an improved compression algorithm, greater colours (8 and 24bit) and a true alpha channel. Again, as it initially was a lossless format, lossy compression was made possible later.

What makes the PNG so powerful is the ability to choose between an 8bit, 256-colour image with transparency, right up to a 16m colour, 24bit-image with an 8bit alpha channel for very photorealistic renderings. This varying bit depth makes the PNG quite a versatile format.

As such, the PNG can be used for a number of things: It can easily replace the GIF, and is able to also provide photorealistic renderings but at the cost of size.

WebP

Although the newest of the bunch, it might be one of the more important formats despite its low usage rate. This Google promoted format possesses much of the features of a PNG, an alpha channel as well as 24bit colour in both lossy and lossless. Much of the advancement in Google’s research has provided significant savings with an aggressive encoder, in comparison to both PNGs and JPEGs. The image below started as a 2.1MB-JPEG, but as a lossy WebP is now just 401KB!



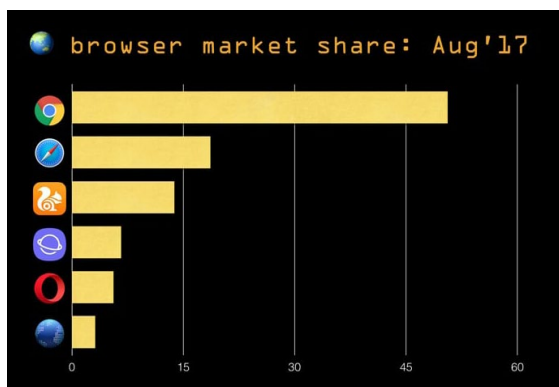
An image showing size as a JPG with its equivalent size as a WebP

In fact, Google believes that the WebP could eventually replace the PNG in the not so distant future.



A Can I Use table indicating WebP browser support

As a Google property, the support came from browsers powered by the Blink rendering engine (above). But since Mozilla announced their impending support for the format [13], it has made the WebP an even more attractive solution considering the data savings.



A chart of browser market share as of August 2017

The browser market share above [14] indicates the need to at least consider this format when possible. In light of just about 50% of your audience is capable of accepting the format, the WebP is a format worth investigating as a developer.

JPEG

The image workhorse of all formats, the JPEG makes up just over 40% of the images used online and is the format of choice by nearly all of the consumer cameras and smartphones worldwide.

Just as with the PNG, making the best use of this format will yield some of the best image optimisations results. The JPEG, present since 1992, has been the focus of plentiful research through the introduction of updated encoders, much of it still on-going.

It was presented as a lossy photographic format, to counter what is a lossless asset in the PNG-24. What resulted was essentially the best and most popular format for anything that needed something beyond 256 colours or photographic and lossy, long before the arrival of the WebP. As such, adoption rate was high. The JPEG eventually also featured lossless variants which, although not used often, are served mostly by CDNs: JPG-2000 (Safari/WebKit supported) and JPG-XR (Microsoft browser supported).

But part of the magic of the JPEG involves something called chroma subsampling.

Chroma subsampling

Chroma subsampling is "*the practice of encoding images by implementing less resolution for chroma information than for luma information, taking advantage of the human visual system's lower acuity for colour differences than for luminance.*" [15]

In plain English, we are working with the human's inability to notice missing colour detail, over light, and by doing so are able to alter some colour data in the image whilst still maintaining fidelity. Essentially, it's a process that averages neighbouring pixel colour data. Why is this important? By performing this action, we are in effect saving even more bytes. The standard YCbCr (chroma subsampling) value where no colour data has been altered is 4:4:4. However, the most data efficient and sought YCbCr value is 4:2:0, where only the light value (Y) remained intact. When complete, the achieved savings can be as much as 75% of the original image size. As such, subsampling is a big part of the JPEG compression.

Remember the oversized image from the car manufacturer? No subsampling had been applied (see below).

```
Henri -- bash -- 89x28
Last login: Mon Nov 27 14:44:31 on tty000
CERN:~ HENRI$ exiftool /Users/HENRI/Desktop/my17_civic_type_r_exterior_gallery_10.jpg
ExifTool Version Number      : 10.56
File Name                    : my17_civic_type_r_exterior_gallery_10.jpg
Directory                   : /Users/HENRI/Desktop
File Size                    : 443 KB
File Modification Date/Time  : 2017:11:27 14:44:21-05:00
File Access Date/Time       : 2017:11:27 14:46:09-05:00
File Inode Change Date/Time  : 2017:11:27 14:46:08-05:00
File Permissions             : rw-----
File Type                   : JPEG
File Type Extension         : jpg
MIME Type                   : image/jpeg
JFIF Version                : 1.01
Resolution Unit             : None
X Resolution                 : 1
Y Resolution                 : 1
Image Width                 : 3840
Image Height                : 2160
Encoding Process            : Progressive DCT, Huffman coding
Bits Per Sample             : 8
Color Components            : 3
YCbCr Subsampling           : YCbCr4:4:4 (1:1)
Image Size                  : 3840x2160
Megapixels                  : 8.3
CERN:~ HENRI$
```

Image of the meta data in an image asset, showing a chroma subsampling of 4:4:4.

Exif Data

Exif data is an area which gets too little attention. No one can really pinpoint where this below photo was taken. This is where exif data comes into play [16].



Image of revellers on a patio

Exif data or the Exchangeable Image File Format is meta data information embedded by digital cameras into a JPG image (in

our case) which will list an abundance of information about the photograph such as (but not limited to): camera model, make, time stamps, altitude, lens information, whether flash was on or off and so much more. In fact, this kind of meta data was identified in over 5,000 distinct fields across various images [17].

This is but a pinch of what's available, but if you look closely (below), you'll be able to spot where GPS information has been included, along with additional information.

```

Setting Method      : One-chip color CMOS
Scene Type         : Directly photographed
Exposure Mode      : Auto
White Balance       : Auto
Focal Length In 35mm Format : 29 mm
Scene Capture Type  : Standard
Lens Info          : 4.15mm f/2.2
Lens Make         : Apple
Lens Model         : iPhone6 Plus back camera 4.15mm f/2.2
GPS Latitude Ref    : North
GPS Longitude Ref   : West
GPS Altitude Ref    : Above Sea Level
GPS Time Stamp      : 2118013
GPS Speed Ref       : km/h
GPS Speed           : 0
GPS Img Direction Ref : Magnetic North
GPS Img Direction    : 127.9089783
GPS Dist Bearing Ref : Magnetic North
GPS Dist Bearing     : 127.9289793
GPS Date Stamp      : 2017-08-24
GPS Horizontal Positioning Error : 16 m
Compression         : JPEG (old-style)
Thumbnail Offset     : 2180
Thumbnail Length     : 11750
Image Width          : 3264
Image Height         : 2448
Encoding Process     : Baseline DCT, Huffman coding
Bits Per Sample      : 8
Color Components     : 3
Y Cb Cr Sub Sampling : YCbCr4:2:0 (2 2)
Aperture            : 2.2
GPS Altitude         : 180.4 m Above Sea Level
GPS Date/Time        : 2017:08:24 23:00:22Z
GPS Latitude         : 42 deg 38' 58.45" N
GPS Longitude        : 79 deg 23' 39.49" W
GPS Position         : 42.6497185, -78.407114, 29 deg 23' 39.49" N
Image Size           : 3264x2448
  
```

Exif data exposing GPS location

This meta data, which is added by your camera and sometimes even your post-processing application like Photoshop, takes up actual space. In a study, an average of 16% of image data was comprised of exif data information [18]. That means that a 100kb image has 16kb of removable data. Now imagine that at scale. This is why exif data should be removed from your images. Most compression applications will do the job.

Photoshop

This is an essential tool for many in editing and exporting images, but we need to be aware of a few details. Apart from needing a slightly more modern JPEG encoder, Photoshop also does the following when working with the said format:

Any JPEG exported image at a quality of 50% or less will have a chroma subsampling value of 4:2:0. Anything else is set to 4:4:4. This means that unless you habitually use images exported at 50% or less, you need to send the image through a third-party app to bring that chroma subsampling back down to 4:2:0 to get the maximum savings. A small detail worth nothing as Photoshop is very much widely used.

Now what?

Now that we understand the moving parts of image optimisations, how do we apply all of this?

First, we make sure that we employ all the tools available at our disposal. Some of the tools are `picture`, `source` and `srcset`. The `picture` element specifically is important in handling different image formats.

`picture`: a responsive images method to control which image resource a user agent presents to a user, based on resolution, media query and/or support for a particular image format.

We would then end up with something like:

```
<picture>
  <source type="image/webp" srcset="optimize_prime.webp">
  <source type="image/vnd.ms-photo" srcset="optimize_prime.jpg">
```

```

</picture>
```

In the above example, anything that supports WebP would load the optimize-prime.webp file and move out. The only moment the optimize-prime.jpg file would be loaded is when none of the user agents is capable of supporting the two previous formats, for example Firefox.

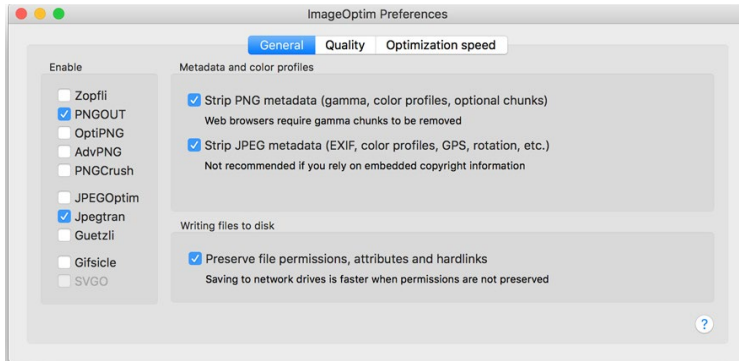
To accommodate breakpoints and responsive design, we can then add some w descriptors which will indicate at which viewport width we would serve a new image.

```
<picture>
  <source type="image/webp"
    srcset="/optimize_prime_100.webp 100w,
    /optimize_prime_400.webp 400w,
    /optimize_prime_800.webp 800w,
    /optimize_prime_1000.webp 1000w,
    /optimize_prime_1200.webp 1200w,
    /optimize_prime_1400.webp 1400w" />
  sizes="(min-width: 500px) 33.3vw, 100vw" />
</picture>
```

The lone challenge with this? You must do the same for each format multiplied by each image you wish to use on a given site. This is possibly where you might want to outsource this image format selection process by using a modern CDN like Cloudinary [19], Imgix [20] or Akamai [21] to do the heavy lifting.

In terms of tools that we need for the optimisation and compression of the images, of the many available on the market, ImageOptim [22] is still one of the best ones. It employs all of the modern encoders in both the case of PNGs and JPEGs, and

is regularly updated, employs SVGO and also strips the metadata right out. It's offered in both a GUI and CLI [23] flavour.



ImageOptim preference panel

Now, to streamline much of this process, task runners such as Gulp [24] will help. There are many options of packages available, including gulp-imagemin [25], which will allow to automate the workflow and will allow for custom plugin options.

```
$ npm install --save-dev gulp-imagemin
Then:
const gulp = require('gulp');
const imagemin = require('gulp-imagemin');

gulp.task('default', () =>
  gulp.src('src/images/*')
    .pipe(imagemin())
    .pipe(gulp.dest('dist/images'))
);
```

Lazy loading

Finally, we need to discuss the idea of lazy loading. As per Wikipedia, *"a design pattern commonly used in computer programming to defer initialization of an object until the point at which it is needed."*

Research has revealed that around two thirds of content on any given web page are below the fold or outside of the viewport. We also have discovered that 50% of users never make it to the bottom of the page [26]. That in mind, we could potentially save bandwidth and data by lazy loading image assets. As such, this a concept that must be kept in mind at all times to reduce wasteful downloads. Lazysizes [27] is one of many techniques that can be used [28].

Testing tools

There are number of very interesting testing tools just for images. I personally like to always keep DevTools open and simply take an early look at the waterfall and page weight, but there are a number of tools that are specific to images that you can use to audit.

- **ResImageLint [29]** — Linter for Responsive Images: this looks at your images and will tell you if they're properly sized or not, giving you a readout of the source and container size delta.
- **NCC Image Checker [30]**: again, to address the need to remind developers how poor resizing can affect the user experience, the NCC offer a Chrome extension to check on the image size.
- **Website Speed Test [31]**: Cloudinary has set up a page to

analyse all images on a page, and offer some simple and raw data about the page weight, and the potential for additional savings.

- **Lighthouse [32]:** we can't mention DevTools without including this auditing tool, which started as a checking tool for progressive web apps. It has now evolved into a fully fledged site auditor, which now provides recommendations on image sizing as well as compression and data savings opportunities. You can find it in the 'Audits' tab of Chrome DevTools.

Conclusion

We have seen the costs associated with poor image management upfront. From eroding user experiences, hampering retail sales, creating bottlenecks and dissolving data plans and even shutting down devices, their management has never been more important. Armed with the discussed tools and techniques, we should all have enough to at least apply the learned choices in our current and upcoming projects, in order to restore the delightful and smooth user experiences we all aim for.

Resources

[1] IDC

Smartphone Volumes Expected to Rebound in 2017 with a Five-Year Growth Rate of 3.8%, Driving Annual Shipments to 1.53 Billion by 2012, According to IDC
<https://www.idc.com/getdoc.jsp?containerId=prUS42334717>

[2] Peter-Paul Koch

Front end and back end
https://www.quirksmode.org/blog/archives/2015/01/front_end_and_b.html

[3] Jiayi Wen & Shengbo Guo, Facebook

News Feed FYI: Showing You Stories That Link to Faster Loading Webpages
<https://newsroom.fb.com/news/2017/08/news-feed-fyi-showing-you-stories-that-link-to-faster-loading-WEBPages/>

[4] Henri Helvetica

The Need for Mobile Speed by DoubleClick
<http://www.afast.site/2016/11/16/the-need-for-mobile-speed-by-doubleclick/>

[5] Addy Osmani, Twitter

<https://twitter.com/addyosmani/status/916391453076602880?lang=en>

[6] Tim Kadlec, London Web Performance

<https://www.youtube.com/watch?v=Wf7d7wQZra8&feature=youtu.be&t=37m54s>

[7] Andy Davies, Twitter

<https://twitter.com/AndyDavies/status/802224884084568065>

[8] Jason Grigsby, Twitter

<https://twitter.com/grigs/status/837026587019128832>

[9] gzip, Wikipedia

<https://en.wikipedia.org/wiki/Gzip>

[10] Brotli, Wikipedia

<https://en.wikipedia.org/wiki/Brotli>

[11] SVG

<https://github.com/svg/svg>

[12] SVGOMG

<https://github.com/jakearchibald/svgomg/blob/master/README.md>

[13] Bugzilla

Implement experimental WebP image support

https://bugzilla.mozilla.org/show_bug.cgi?id=1294490

[14] Statcounter

Mobile Browser Market Share Worldwide

<http://gs.statcounter.com/browser-market-share/mobile/worldwide>

[15] Chroma subsampling, Wikipedia

https://en.wikipedia.org/wiki/Chroma_subsampling

[16] Exif, Wikipedia

<https://en.wikipedia.org/wiki/Exif>

[17] Kalev Leetaru, Forbes

The Hidden World Of News Imagery EXIF Metadata & iPhones As News Cameras

<https://www.forbes.com/sites/kalevleetaru/2016/10/19/the-hidden-world-of-news-imagery-exif-metadata-iphones-as-news-cameras/>

[18] Inian Parameshwaran, Dexecure

Impact of metadata on Image Performance

<https://dexecure.com/blog/impact-of-metadata-on-image-performance/>

[19] Cloudinary

<https://cloudinary.com/>

[20] Imgix

<https://www.imgix.com/>

[21] Akamai

<https://www.akamai.com/>

[22] ImageOptim

<https://imageoptim.com/api>

[23] ImageOptim-CLI

<https://jamiemason.github.io/ImageOptim-CLI/>

[24] Gulp

<https://gulpjs.com/>

[25] gulp-imagemin

<https://www.npmjs.com/package/gulp-imagemin>

[26] Zoltán Kollin, UX Myths

Myth #1: People read on the web

<https://uxmyths.com/post/647473628/myth-people-read-on-the-web>

[27] lazysizes

<https://github.com/aFarkas/lazysizes>

[28] Maria Antonietta Perna, SiteGround

Five Techniques to Lazy Load Images for Website Performance

<https://www.sitepoint.com/five-techniques-lazy-load-images-website-performance/>

[29] RespImageLint

<https://github.com/ausi/respimagelint#respimagelint---linter-for-responsive-images>

[30] NCC Image Checker

<https://chrome.google.com/webstore/detail/ncc-image-checker/fphiheficgnpphmdliclanppccfgifl>

[31] Website Speed Test

<https://webspeedtest.cloudinary.com/>

[32] Lighthouse

<https://developers.google.com/web/tools/lighthouse/>

MAKE YOUR ANIMATIONS PERFORM WELL



The author
Anna Migas

Anna Migas (twitter.com/szynszyliszys) has been working as a front-end developer and designer for over five years. During her time at Lunar Logic (lunarlogic.io), she's had a chance to create and maintain a few open source projects related to web animations – for example, Starability (blog.lunarlogic.io/starability). This experience helped her understand animation performance in detail. In her spare time, Anna loves skateboarding, reading books, and travelling.

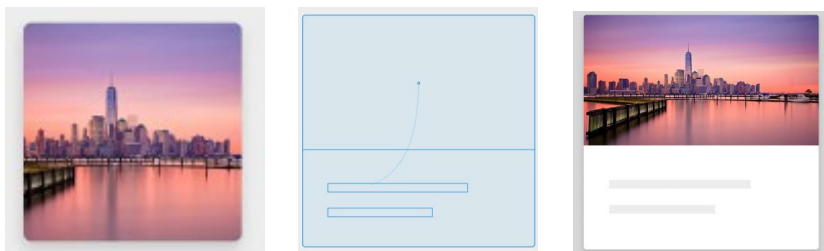
Over the last few years web animations have gained in popularity. This was caused both by the access to a new technology (CSS animations and transitions are now supported in most browsers) and emerging design trends. When executed properly, an animation can make our website or app more attractive and engaging to a user. When executed poorly, it can be deadly to our website's performance or make the user wonder why it was even added. The app can seem slow, non-responsive or inconsistent.

Animation should be an enhancement that makes it easier for the user to understand how our website works. For example, an animation that supports user interaction can be a good idea. Adding an animated check mark icon can be a great way to give the user feedback that the action they wanted to perform was successful. Or you could animate the transitions between views to help the user understand how the website's navigation works.



A check mark icon that slides into the view can be an indication of a successful interaction

A bad example would be an animated slideshow that is auto-played: most users tend to ignore images showcased this way. They are in many ways similar to ads, and we are forcing users to download image assets just for this purpose. We also need to remember that some users have trouble focusing their attention when there is movement on the screen, so adding a slideshow can make it harder for them to browse the content of a website. Having said that, the first advice on animation performance I am going to give you is to **not use animation for the sake of animating**. Every time you decide to add an animation, make sure it really enhances the user experience of your website or app.



An example of an animation that helps the user understand how to navigate the website [1]

Choosing the right tool

Once you set your heart on adding an animation to your project, you need to choose the right tool. Some people say that CSS is better for animating than JavaScript (or vice versa), but it just depends on your needs. Both CSS and JavaScript are great tools for animating, but each technology has its pros and cons.

If you need to animate something really simple (for example, an element entering a screen), CSS is the way to go. CSS animations (and transitions) are native to the browser and you don't need to download any external assets to use them. Browsers have additional optimisations implemented for CSS animations. A browser runs most of its tasks in something called the "main thread". At the same time, the majority of CSS animations are being run in a separate thread called the "compositor thread". This means that while the CSS animation is running, it doesn't interrupt or delay other tasks that the browser needs to take care of. This is a great advantage of CSS animations. However, it's hard to forget about their biggest disadvantage: scheduling two or more CSS animations together is difficult and inconvenient to work with.

This is where JavaScript animations shine: working with multiple animations that need to be tied together is seamless. There are

many ways to animate things with JavaScript. The ones worth mentioning are:

- Pure JavaScript
- Web Animations API [2]
- A framework (for example, the GreenSock Animation Platform (GSAP) [3] or anime.js [4])

There are arguments for and against each method. Pure JavaScript needs additional optimisations to be performant (see the `requestAnimationFrame()` function described later in the article) but is lightweight and in many cases can be a good solution.

The Web Animations API is the best of both worlds, as it's native to the browser, gets the same browser optimisations as CSS animations and still gives you control over scheduling and timing multiple animations. Unfortunately, the support for the Web Animations API is still limited (Chrome 36+, Firefox 48+ and Opera 29+ have basic support), and it needs a polyfill to work everywhere properly. Once native support for the Web Animations API is more common, it could become the most sensible choice of all.

If your website relies heavily on animations, you might consider using a framework. At the moment the GreenSock Animation Platform (GSAP) is the recommended solution, as all the required optimisations are implemented. However, it's quite heavy. Anime.js is a bit more limited but a lighter solution that you might want to consider.

In conclusion: CSS is best for simple animations, while JavaScript is a great alternative when you need to have more control over multiple animations' scheduling and timing.

Understanding the rendering process

Before we start optimising, we need to understand how a browser renders a website. It will tell us how our animation can affect the whole experience and which properties can be animated in a performant way. So the steps a browser needs to take are:

1. Send a request to a server.
2. Create DOM elements (all HTML elements).
3. Recalculate the styles.
4. Calculate the layout (sizes of elements).
5. Paint (layer creation).
6. Compositing (putting layers together for a frame).



Steps a browser needs to take to show a page to a user

This is quite a lot of work to show a website to a user! Also, if any change (for example, scrolling down or triggering an animation) is introduced by the user, the browser needs to go through all these steps again. The only difference here is that instead of the first two steps (request and DOM creation) there can be a JavaScript trigger at the beginning of the flow.

There are three types of changes that can be introduced, and some of them are more likely to cause jank (website stuttering) than others:

1. Layout change

The most costly change. It makes the browser go through all the steps during the rendering process, because we alter the geometry of the page. It's triggered when we animate properties such as margin, top, width, display, etc. We should avoid animating these properties as well as animations that cause other elements to move with them.



A change of layout triggers all browser rendering steps

2. Paint change

In this case the browser knows exactly how much space each element covers, so it can skip the layout step in the rendering process. Paint changes when we animate background, box-shadow, color properties and similar. A less costly change, but still can cause problems on some devices or if we animate too many elements at once.



A paint change skips the "layout" rendering step

3. Compositing change

The change that happens when the browser already knows which element is on which layer (created in the Paint stage) and only needs to put the layers together. This is the change that is most performant as some browsers can use a device's GPU to draw the image to the screen at this point. It happens as we animate transforms (translate, rotate, scale) and opacity. That

said, transform and opacity are the properties that are the best for animating. They give us enough options to create almost any type of animated content and we should stick to them.

JavaScript

Recalculate Styles

Layout

Paint

Composite Layers

A compositing change is likely to omit the "layout" and "paint" rendering steps

If you want to quickly check what type of change is bound to happen for any property, visit csstriggers.com [5].

Optimisation techniques

Now that we know we should focus on animating only transforms and opacity, we can start the real optimisations. There are three techniques that we will cover in this section: the will-change property, the FLIP principle and the `requestAnimationFrame()` function.

The will-change property

As mentioned before, during the Paint phase the browser puts elements on layers. Layers in the browser could be compared to layers in Photoshop: if two things are on the same layer, they are glued together and there is not much we can do with them. If elements are on separate layers, they can be moved around and don't affect other layers. We can take advantage of that and put elements on different layers to animate them without causing browser repaints (making the browser go through the Paint step again).

By default, the browser puts all elements on one layer. There are certain circumstances that "promote" elements to the new layers (it can vary slightly in different browsers):

- 3D or perspective transforms
- Animated 2D transforms or opacity
- Being on top/a child of an existing layer
- Accelerated CSS filters
- In special cases <video>, <canvas>, plugins
- The will-change property

Let's look at an example to understand why the will-change property can be useful. If we want a button to move slightly up and down on hover, we could write the code like this (see over:

```
@keyframes move-up-down {

  0% {
    transform: translateY(0);
  }

  50% {
    transform: translateY(-10px);
  }

  100% {
    transform: translateY(0);
  }
}

button {
  background-color: white;
  border: 1px solid blue;
}

button:hover {
  animation: move-up-down 1s infinite;
}
```

The problem is, the browser doesn't yet know that the element will be animated with the 2D transform property. By default both button and all other elements around it will be on the same layer. When a user hovers over the button, the browser will repaint the whole page to put the button on another compositing layer and the user can experience a slight lag before the element moves.

What is worse, when they move the cursor away, the browser will repaint the page again to put elements back on the same layer. This may cause performance flaws when our website is complex.

However, if we use the `will-change` property ahead of the interaction, the browser will have separate layers for the button and its surroundings and the animation will not cause any browser repaints:

```
@keyframes move-up-down {

  0% {
    transform: translateY(0);
  }

  50% {
    transform: translateY(-10px);
  }

  100% {
    transform: translateY(0);
  }
}

button {
  background-color: white;
  border: 1px solid blue;
  will-change: transform;
}

button:hover {
  animation: move-up-down 1s infinite;
}
```

There are a few things we need to remember when we decide to use the will-change property:

- Use it **before the animation** or transition is about to start. If we add will-change at the same time as the browser is supposed to start animating an element, it can have a negative effect. At this point the browser is already busy trying to create the new layer and the will-change property would be yet another thing to take care of.
- Don't overuse it — each new layer consumes the device's memory. If we put too many elements on separate layers, the browser can crash.
- Use it in stylesheets if an animation/transition is bound to be triggered many times. If we want to animate something only once, a better option would be to add will-change with JavaScript and remove once the movement is finished:

```
var element = document.getElementById('element');
    element.addEventListener('mouseenter', hintBrowser);
        element.addEventListener('animationEnd', removeHint);

function hintBrowser() {
    this.style.willChange = 'transform';
}

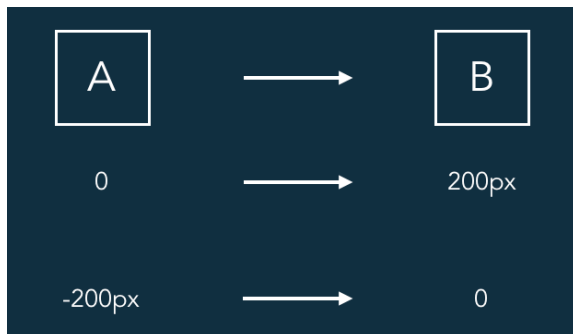
function removeHint() {
    this.style.willChange = 'auto';
}
```

It's not supported in Internet Explorer/ Edge but you can use a 3D transform hack instead:

```
-webkit-transform: translate3d(0,0,0);
```

The FLIP technique

FLIP stands for First Last Invert Play. It's a principle that was first introduced by Paul Lewis [6], a developer advocate at Google. We take the first and the last position of an animated element and invert them before playing the animation. So if we wanted to do a simple movement of an element by 200 pixels to the right, we would first pull the element 200px to the left and then let it slide to its final position:



An inverse way of thinking about changing element's position is a core of FLIP technique


```
@keyframes slide-from-left-flip {
  0% {
    transform: translateX(-200px);
  }

  100% {
    transform: none;
  }
}
```

Why should we even do that? Even though we are using performant transforms for the movement, without FLIP the browser still needs to take a moment to calculate the final position of the element once the animation is triggered. This can be perceived as a small lag between the time the user interacts with the element and when the animation is played. If we use FLIP, the browser makes this calculation before the user even has a chance to use the page.

There is a gap of around 100 milliseconds before a person is able to interact in any way with the rendered page — our brains are just not fast enough to do it earlier. And we are using this 100ms gap to calculate the positions of elements — if the final position is already known, the browser has all the work done before and just lets the element move to the point from where it started. Thanks to FLIP, when the user triggers the element, it feels like the animation is played instantly. It can make a difference for users with less powerful devices (for example, low-end smartphones). FLIP should be used only for the interactions that happen on user input.

If you decide to animate a more complicated scenario than just moving an element from one side to another, I recommend using FLIP.js [7] — a helper library to make calculating the final state of elements easier. To understand the principle fully, read the blog post written by its creator, Paul Lewis [8].

The `requestAnimationFrame()` function

If you are using CSS, GSAP, the Web Animations API or even jQuery 3.0.0+ for your animations, you have nothing to worry about. But if you, for example, decide to animate elements with pure JavaScript, you should familiarise yourself with the `requestAnimationFrame()` function.

Earlier in the article we went through the steps that the browser needs to take before showing a website to the user. These steps form what we call a “frame”. To create an animation that looks smooth, a browser needs to draw around 60 frames every second. A quick calculation ($1000\text{ms}/60 = 16.666\text{ms}$) shows us that the browser has only 16ms to build each frame. If the browser gets interrupted (for example, by JavaScript changing the size of an element), it needs to go through all the steps again. In this case, the browser is likely to not create the frame in time and miss its 16ms deadline. Unfortunately, this will be visible as jank — the animation is stuttering.

We can prevent this by using the `requestAnimationFrame()` function. It ensures that all the required JavaScript code is scheduled at the earliest possible moment for each frame and the browser has enough time to apply the changes. The frames are not queued for the future and drawn only when the browser is ready to paint them to the screen — this prevents unnecessary draws. Another advantage is the fact that the browser doesn't play animations if the tab is not visible — a way to save memory resources and battery life. Using the function is quite easy:

```
function repeated() {  
    // show something many times  
    window.requestAnimationFrame(repeated);  
}  
window.requestAnimationFrame(repeated);
```

Once we want the animation to stop playing, we can call the complementary function, `cancelAnimationFrame()`:

```
window.cancelAnimationFrame(repeated);
```

The function is supported in all modern browsers at the moment. If you want to support older browser versions, you need to use `requestAnimationFrame()` with vendor prefixes. To support IE9 and older, use this polyfill [9] (which takes care of vendor prefixes by default).

Testing your animations

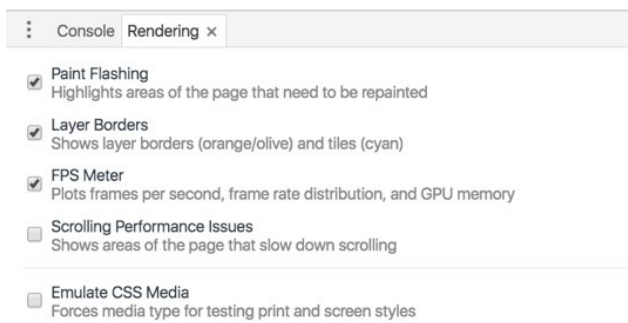
It might seem obvious, but we should always test our animations. Even thoughtful and optimised code can cause jank. At the same time, I have seen examples of animations which seem to work seamlessly, even though their code was full of imperfections.

The best way to test an animation is to use developer tools. I personally use both Chrome and Firefox to get the accurate measurements. I will quickly go through the options that are most useful for animations.

Chrome

Rendering tab: When enabled (in More tools > Rendering), it allows us to toggle a few options:

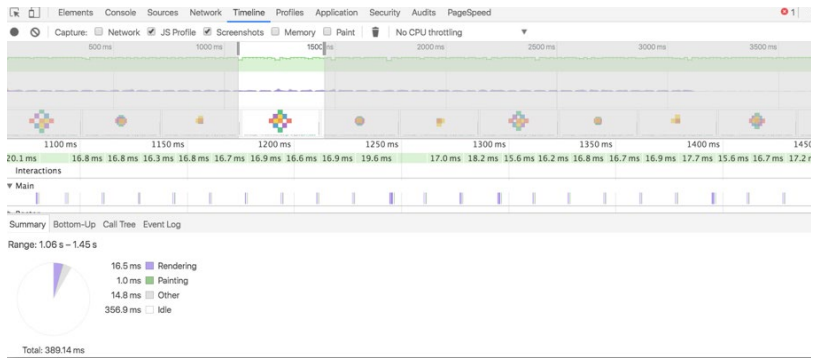
- **Paint Flashing:** helps to determine which elements get repainted
- **Layer Borders:** all layers are shown with a border
- **FPS Meter:** presents us with a number of frames per second metric
- **Scrolling Performance Issues:** shows us if an element slows down the scrolling performance.



Animations tab: Added recently, makes it possible to play animations at a slower rate (25% or 10%), pause it or show certain frames.

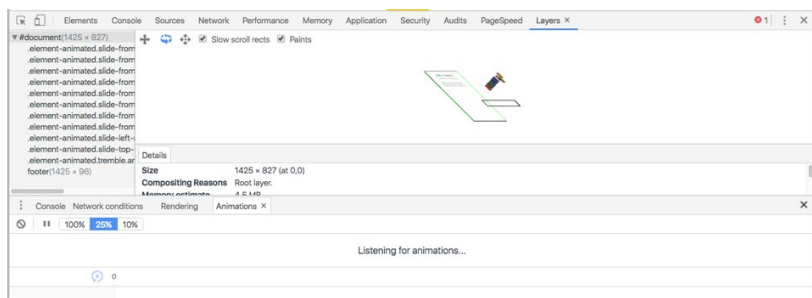
Performance tab: Lets us record the screen and any changes with a precision of up to one millisecond (the shorter the recording the easier it is to debug). We can see screenshots of how the animation develops over time, the length of frames and the summary of steps

Chrome had to take to draw each one.



A way of testing animations — using the “Performance” tab in Google Chrome

Layers tab: A new addition, shows us why an element (or group of elements) was put on the new layer. We can play with options and even see how our layers look in 3D view when animated (although it can be slow at times).



The “Layers” and “Animations” tabs enabled in Google Chrome

Firefox

Animations tab: Similar to the one in Chrome. We can see the animations in slow motion and stop them at any given point by using the draggable widget. A nice addition is an indication whether our animation is optimised (a small lightning bolt icon) and a tooltip with additional information.

Summary

I hope that reading this article has helped you understand the rules of animation performance and inspired you to test your own creations to avoid performance flaws. Let's sum up the most important points:

1. Don't use animations for the sake of animating. Use them to help users achieve their goal.
2. Animate only transform and opacity properties.
3. Use the will-change property, the `requestAnimationFrame()` function and the FLIP technique when applicable.
4. Avoid creating too many layers.
5. Animate elements that are on the top layers (not hidden below other elements).
6. Test animations before optimising.

Further reading

If you are interested in learning even more, make sure to check out these resources:

Jank Free [10]: contains articles on the causes of jank and ways to prevent it

High Performance Animations [11]: an article to get a deeper understanding on why only transitions and opacity should be animated

Web Fundamentals — Design & User Experience: Animations [12]: more information on developing animations by the Google team

Resources

[1] Material.io

How does material move?

<https://material.io/guidelines/motion/material-motion.html#material-motion-how-does-material-move>

[2] W3C

Web Animations

<https://www.w3.org/TR/web-animations-1/>

[3] Green Sock

<https://greensock.com/>

[4] anime.js

<http://animejs.com/>

[5] CSS Triggers

<https://csstriggers.com/>

[6] Paul Lewis

<https://aerotwist.com/>

[7] FLIPjs

<https://github.com/googlearchive/flipjs>

[8] Paul Lewis

FLIP Your Animations

<https://aerotwist.com/blog/flip-your-animations/>

[9] rAF.js

<https://gist.github.com/paulirish/1579671>

[10] Jank Free

<http://jankfree.org/>

[11] Paul Lewis & Paul Irish, HTML5 Rocks

High Performance Animations

<https://www.html5rocks.com/en/tutorials/speed/high-performance-animations/>

[12] Paul Lewis, Google Web Fundamentals

Animations

<https://developers.google.com/web/fundamentals/design-and-ux/animations/>

PERFORMANT WEB FONT TECHNIQUES



The author
Matt James

Matt James (mattjamesmedia.com) is a front-end developer from St. Louis, Missouri, focused on crafting performant, device agnostic, and accessible experiences. He followed a long road through photography, the cycling industry, personal training, and freelance web design to arrive at a place where he is consistently trying to increase the quality of web offerings for his organisation. With a near-daily influx of new frameworks and libraries, Matt tries to stay focused on which techniques will ultimately lead to better user experience while still maintaining solid baseline interactions for all users, regardless of their browsing capabilities.

When web fonts first appeared in 2009, they brought with them the promise of rich typographic control, unique branding and more leverage for web designers to carve out an identity for their clients. Gone were the days of relying on web safe fonts, when designers were resigned to the likelihood that a generic font would end up in their carefully crafted layout.

However, for all their added polish, web fonts present a unique problem for web performance. Despite accounting for a relatively small slice of the average payload, web fonts have a unique ability to bring the whole experience to a halt. A browser's default behaviour, when it encounters a web font, is to prevent any text from rendering until the fonts have loaded and parsed, causing a flash of invisible text or FOIT. The majority of browsers will wait up to three seconds before giving up and providing a fallback font. However, older WebKit browsers have an unlimited FOIT period while waiting for a font. This means that for users on iPhones that haven't updated for a few years, web fonts can be a single point of failure in an app rendering.

It doesn't have to be like this. If we look at web fonts as a progressive enhancement, a nice-to-have in ideal conditions, and aim to implement some font-loading strategies to counter the render-blocking nature of web fonts, we can have both the rich typographic identity that web fonts afford and a reliable, performant experience that ensures our users get content regardless of network conditions. By embracing FOUT, or a flash of unstyled text, as the default, content is served with a fallback font and doesn't have to rely on the availability of a web font to render.

Fortunately, the browser landscape is beginning to offer developers a number of native tools to optimise font-loading performance. While browser support is still evolving for a number of these APIs, there are several reliable and robust polyfills that

help fill in the gaps for non-supporting browsers. Let's dig in and take a look at some of those native features first.

The font-display property

One of the most exciting recent developments in the CSS spec is the font-display descriptor that can be included in the @font-face rule. This new descriptor is a boon to web developers because it can, in one line of CSS, declare to the browser exactly how it should handle the period of time during which the custom font is loaded.

Using font-display is simple. Just drop it inside the @font-face rule and feed it one of five potential values to specify the intended font-loading behaviour.

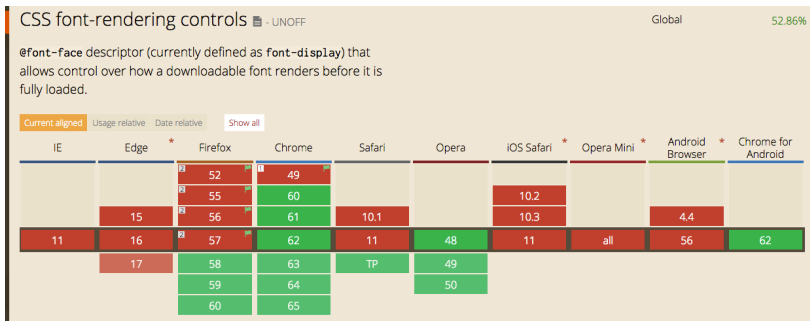
- auto just tells the browser to do what it would normally do with web fonts. As mentioned before, this usually means a three-second period of render-blocking before a fallback font is rendered.
- block explicitly instructs the browser to block rendering of the text until the web font is loaded. From a performance perspective, this doesn't provide any real benefit to the user, but it does ensure that a custom font will be the first font users see when that design element is of critical importance.
- swap allows for no render-blocking period and provides a fallback font immediately while the custom font loads. The custom font is then allowed to replace the fallback font whenever it loads, whether that's 500ms or 10 seconds. This is great because it loads text immediately, no matter what, but could be jarring when the custom

font takes a long time to load, forcing a repaint of the content.

- fallback acts as a bit of a compromise between block and swap with a block period of 100ms for the web font to load before a fallback font is rendered. Then, if the web font loads in under three seconds, the fallback will be replaced. Otherwise it will continue to be shown.
- optional immediately renders the fallback font and gives the web font 100ms to download. If it doesn't load in that time, it won't be rendered on the initial visit. However, it keeps loading in the background and can be cached, either in the browser cache or a Service Worker cache to be used in subsequent visits.

When added to a standard `@font-face` block, `font-display` looks like this:

```
@font-face {  
  font-family: 'Montserrat';  
  src: url("/font/Montserrat-Regular.woff") format("woff");  
  font-weight: 400;  
  font-style: normal;  
  font-display: swap;  
}
```



With support for the font-display descriptor on the rise, developers will soon be able to dramatically improve their font-loading performance with one line of CSS

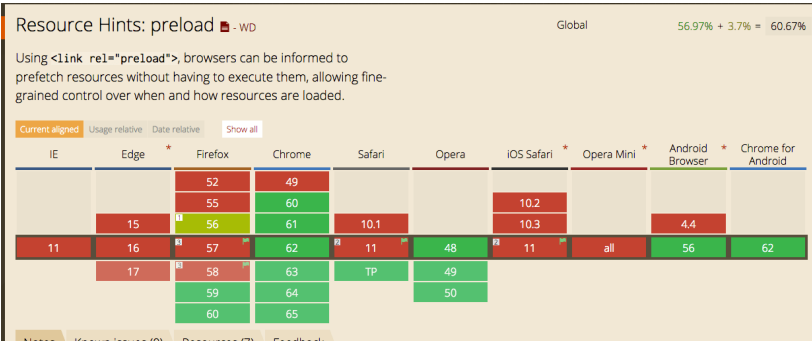
Preload

When combined with another new browser feature, `rel="preload"`, `font-display` can almost completely eliminate the performance hit that comes from web fonts. The benefit of adding the `rel="preload"` attribute is that it tells the browser that these resources are of high priority and should have priority in loading. If it's applied to font files, those fonts are available much earlier in the rendering process, minimising the amount of time to load. Combined with a judicious use of `font-display`, these two browser features could virtually eliminate FOUT.

```
<head>
  <link rel="preload" href="/font/Montserrat-Regular.woff2"
as="font" type="font/woff2" crossorigin>
</head>
```

While `font-display` and `rel="preload"` will dramatically simplify performant font-loading strategies in the future, their current

browser support is limited to Chrome and Chromium-based browsers. While Safari and Firefox have both font-display and rel="preload" support in the works, the current support matrix means that we need to look at more complex JavaScript-based solutions to cover a wider user base.



As more browsers implement rel="preload", fonts can be loaded much earlier in the rendering process, reducing their impact on performance

The Font Loading API

One such solution is the native Font Loading API. The Font Loading API is a new addition to CSS and provides a JavaScript API to exert explicit control over the behaviours of @font-face rules. One of its major benefits is the ability to programmatically create new @font-face rules, meaning that fonts can load asynchronously, preventing their default blocking behaviour. This also eliminates the need to manually write @font-face rules in your CSS.

The FontFace constructor takes three arguments when instantiated: the family name, the src information, and — optionally — an object containing the descriptors typically associated with @

font-face rules. Setting up an instance of Montserrat using the Font Loading API would look like this:

```
var montserrat = new FontFace('Montserrat', 'url(Montserrat-Regular.woff2) format("woff2"), url(Montserrat-Regular.woff) format("woff")', {
  weight: 'bold',
  style: 'normal'
});
```

This is all well and good, but once a new instance of a `FontFace` has been created, its `load` method needs to be called to add it to the page and make use of it. The `load` method returns a promise that resolves when the font successfully loads and rejects if the font fails to load. Tapping into the `load` method, the newly created Montserrat `FontFace` can be added like so:

```
montserrat.load().then(function() {
  document.fonts.add(montserrat)
})
```

To use multiple font files with the Font Loading API, there are a couple potential paths to take. First, you can create each one individually and load them separately. This ensures that one font is not dependent on another font having loaded for it to render.

However, there are times when you might want to load you web fonts together if they are related or you want to avoid multiple reflows of your text if the fonts load at different times. To do this, make use of `Promise.all` and pass it an array of load calls on each font.

```

    var montserrat = new FontFace('Montserrat', url(Montserrat-Regular.
woff) /*other FontFace Info*/);

    var merriweather = new FontFace('Merriweather', url(merriweather.
woff) /*other FontFace Info*/);

    Promise.all([
        montserrat.load(),
        merriweather.load()
    ]).then(function() {
        document.fonts.add(montserrat);
        document.fonts.add(merriweather);
    });

```

Font Face Observer

As powerful as the control afforded by the Font Loading API is, it, too, is a bit limited in its browser support. Currently, it's supported by the latest versions of Chrome and Chromium-based browsers, Firefox and Safari. Unfortunately, Microsoft Edge or older browser versions do not support it. Fortunately, there is a polyfill, Font Face Observer [1], that aims to replicate the Font Loading API and extend its behaviour to a much broader collection of browsers.

The basic syntax of Font Face Observer looks similar to that of the Font Loading API. Despite its similarities, there are a couple key differences in terms of behaviour. First and foremost, Font Face Observer expects an existing `@font-face` rule. Because of this, Font Face Observer does not take a list of URLs as its first argument. Instead, it takes the font-family name referenced in the `@font-face` rule. By default, this means that the web fonts are no longer asynchronous and will block render until loaded. However, Font Face Observer returns a promise, which resolves when the fonts

have loaded. Using this behaviour, a CSS class can be added to the `<html>` element, which can then be leveraged to apply the font to the page.

To use Font Face Observer, you could include a link to it in the `<head>` or include the script within an existing script file. Once the script is available, you then instantiate a new `FontFaceObserver` to monitor the loading of a given font. While this is certainly an effective way to leverage the polyfill, you can squeeze a bit more performance by dynamically creating it in the `<head>` of your file and utilising the `onload` event to trigger your code.

```
<head>
<script>
var ffo = document.createElement('script');
ffo.src="/js/fontfaceobserver.js";
ffo.async = "true";
ffo.onload = function() {
    //Font Face Observer loading code
};
document.head.appendChild(ffo);
</script>
```

Looking back at the use of Montserrat, loading it with Font Face Observer would look like this:

```
var montserrat = new FontFaceObserver('Montserrat');
montserrat.load().then(function() {
    document.documentElement.className += " fonts-loaded";
});
```

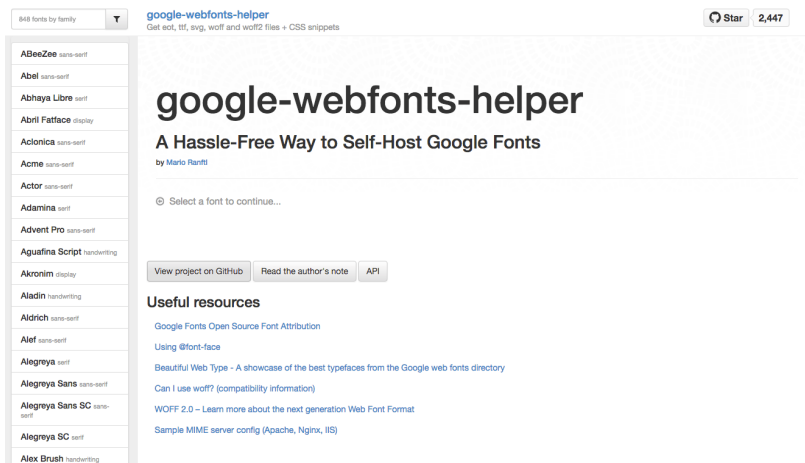
Then, within the CSS, reference the class that has been added to the `<html>` element to apply the web fonts.

```
p {  
    font-family: Helvetica, sans-serif;  
}  
.fonts-loaded p {  
    font-family: 'montserrat', Helvetica, sans-serif;  
}
```

Font Face Observer provides a consistent experience across browsers on its own and has the added benefit of being useful with web font services like Google Fonts. This basic implementation means a user will always see the fallback font first and experience the web fonts once they have loaded. This can be optimised further by caching the fonts after the initial visit, minimising the period of FOUT.

Web font services

So far, with the exception of Font Face Observer, these techniques have primarily focused on self-hosted fonts, which give you far more control over the implementation. However, many developers rely on a font service like Google Fonts or Typekit where the source files are either abstracted away or have restrictions due to licensing agreements. For services with restrictions, there is still a lot that can be done with browser caching and Font Face Observer, but there are new tools developing that will allow for the self-hosting of fonts from Google.



Google Web Font Helper gives developers the ability to download font files typically served by Google, allowing for more control over their font-loading strategies


Google webfonts helper [2] is an open source project that enables a user to download font files that are traditionally hosted on Google. This gives developers more freedom to add font-display — use rel=“preload” or implement the Font Loading API. The increase in control can lead to a major increase in perceived performance. One downside is the loss of the Google CDN network, but leveraging a CDN for all your static assets can circumvent that.

Avoid a jarring shift in layout

All of the techniques mentioned introduce at least some amount of FOUT to the page. While this is optimal for content delivery, it can leave something to be desired from a design standpoint. Fortunately, tools like Font Style Matcher [3] can provide styling options to better set up a fallback font, ensuring a less jarring experience for the user when the text reflows.

Font style matcher

If you're using a web font, you're bound to see a flash of unstyled text (or FOUT), between the initial render of your websafe font and the webfont that you've chosen. This usually results in a jarring shift in layout, due to sizing discrepancies between the two fonts. To minimize this discrepancy, you can try to match the fallback font and the intended webfont's x-heights and widths [1]. This tool helps you do *exactly* that.

Fallback font Georgia	Web font Merriweather
	<input checked="" type="checkbox"/> Download from Google Fonts
Font size: 16px	Font size: 16px
Line height: 1	Line height: 1
Font weight: 300	Font weight: 300
Letter spacing: 0px	Letter spacing: 0px
Word spacing: 0px	Word spacing: 0px
 Copy CSS to clipboard	 Copy CSS to clipboard
The fox jumped over the lazy dog, the scoundrel.	The fox jumped over the lazy dog, the scoundrel.

Font Style Matcher provides a simple interface to create more accurate fallback font styling to reduce the potentially jarring effect that can happen when a web font replaces the system font

Conclusion

As you can see, for all the rich design options afforded by custom web fonts, they do have the potential to introduce a good amount of complexity for developers looking to ensure efficient content delivery. This is complicated even further for each font file specified in a design. Fortunately, as browser support expands for font-display and rel="preload", font loading will become increasingly simple. And for designers who want to really leverage typography with multiple font styles and font weights, the upcoming implementation of variable fonts — fonts that have all the information for every conceivable variant built into one file — means that users will only need one or two font files, and

designers will be able to expand their typographic palette.

With all these new browser APIs and advanced font-loading techniques, developers and browsers alike have finally addressed the performance issues initially introduced by web fonts. As time goes on and the typographic ecosystem continues to mature on the web, both designers and developers will have more control over type implementation than ever before and users will never again have to feel the stinging pain of invisible text while waiting for a custom font.

Further resources

Bram Stein's *Webfont Handbook* [4] from A Book Apart is the go-to guide for all things web fonts. It covers everything from implementation and behaviour to performance and the future of web fonts in a concise but thorough package.

Zach Leatherman's Comprehensive Guide to Font Loading Strategies [5] details a wide variety of font-loading techniques designed to optimise the performance of text rendering. The techniques range in complexity, but each has a specific use case that can be applied to a given application.

The Web Font Optimization page by Ilya Grigorik [6] on Google's Web Fundamentals site offers a comprehensive round-up of default font behaviour and further recommendations for performance optimisation.

Resources

[1] Font Face Observer

<https://fontfaceobserver.com/>

[2] google-webfonts-helper

<https://google-webfonts-helper.herokuapp.com/fonts>

[3] Font style matcher

<https://meowni.ca/font-style-matcher/>

[4] Bram Stein, A Book Apart

Webfont Handbook

<https://abookapart.com/products/webfont-handbook>

[5] Zach Leatherman

A Comprehensive Guide to Font Loading Strategies

<https://www.zachleat.com/web/comprehensive-webfonts/>

[6] Ilya Grigorik, Google Web Fundamentals

Web Font Optimization

<https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/webfont-optimization>

MEASURING PERFORMANCE



The author
Andy Davis

Andy Davies (twitter.com/andydavies) is the associate director for web performance at NCC Group, where he helps clients to understand and improve the performance of their websites. He regularly speaks about performance, is the co-author of *Using WebPageTest*, and the author of *The Pocket Guide to Web Performance*, both available at andydavies.me/books.

To improve the performance of our sites, we need to understand how fast or slow they currently are and in this article we'll explore the three main approaches to measuring performance.

We'll look at the problems each one is good at solving, which should give you a good starting point to explore further and find the tools that work best for your needs.

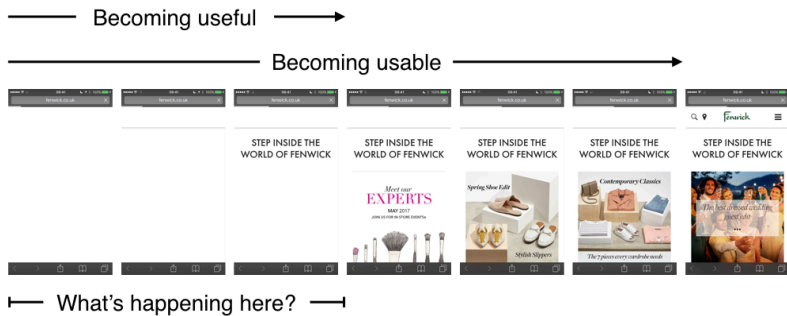
What to measure

When we think about performance, it's easy to think of in terms of page size or number of requests but they're not really measures of visitor experience — I've seen huge pages that were fast, and small pages that were slow.

I prefer to approach performance from a timing view — how long does the visitor wait for a page to start display, when is it visually complete, when can they start to use it?

Depending on the type of site some of these timings are more important than others. For example, on a news site it's 'when can someone start to read the content', for a retail site it might be 'when can someone see the alternative views of a product, add it to their basket etc'.

Measuring these events is key to understanding and improving actual performance.



The home page for department store Fenwick loading on a mobile phone

There are other tools such as YSlow [1], PageSpeed Insights [2] and Yellow Labs [3] that score performance and give some advice on how to improve it.

There's a beautiful simplicity to scores, but they're detached from the events visitors care about, so we'll concentrate on tools and approaches that allow us to time these events.

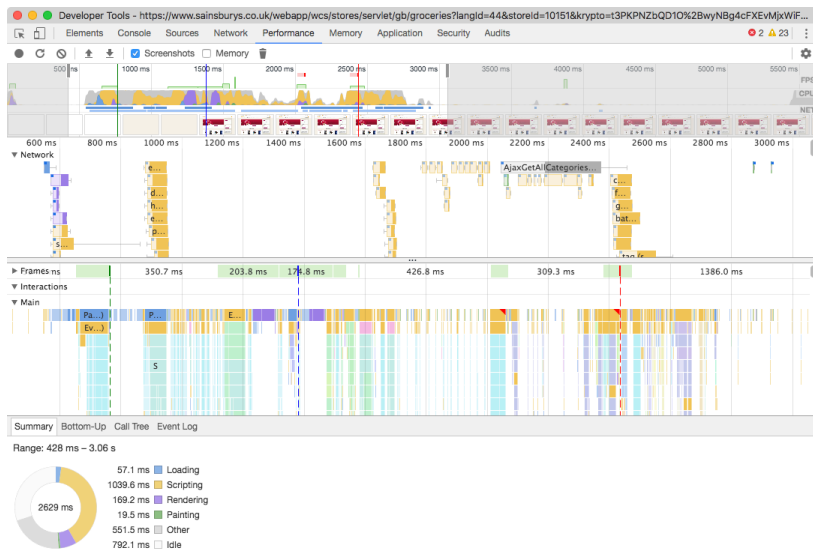
How to measure performance

Developer tools

Browser developer tools are often the place most of us begin to think about performance. They're close at hand, feature rich and provide an incredible depth of information.

They have waterfalls that show the timings of the network requests, flame charts that enable us to get a deep view on what JavaScript functions are being called and how long they take to execute and timelines to tie all the data together in a single view.

The level of detail they provide allows us to dig into what the browser is doing and how the structure of our pages affect how quickly they start to show content, become interactive etc.

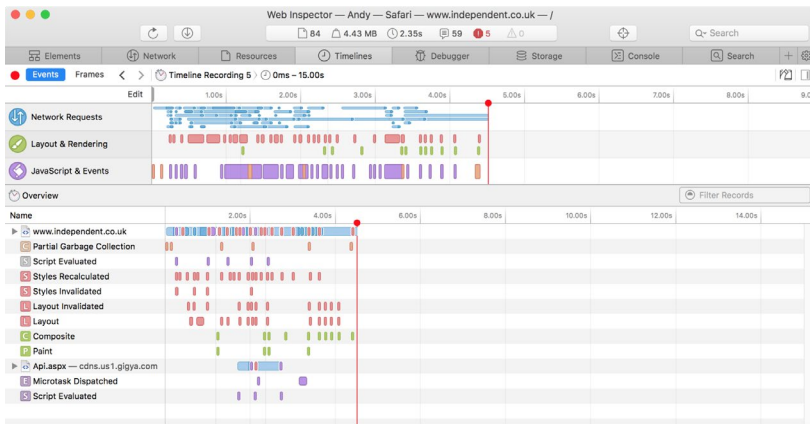


Chrome DevTools Performance Timeline

In Chrome's performance timeline view we can line up what's being displayed on the screen with the internal activity of the browser — the resources being loaded and how long they take to parse and be executed (in the case of JavaScript).

It also allows us to view a subset of the timeline, so we can focus on just what occurs before the page starts to render, for example, and see when the HTML stops being parsed while scripts execute.

We can also tether a mobile phone to our desktop to measure and debug performance directly on the device.



Inspecting the performance of a page in Safari Mobile

This level of detail allows us to diagnose where our performance bottlenecks are, make changes and then see how performance has improved. But as well as diagnostic capabilities we often want to track how our site's performance changes over time and present that in ways that are understandable for everyone.

Synthetic testing

Browser developer tools are great for doing a deep dive into the performance of a page or site and with the advent of headless browsers, which don't have a traditional user interface and can be controlled via test scripts, they can be incorporated into automated performance testing.

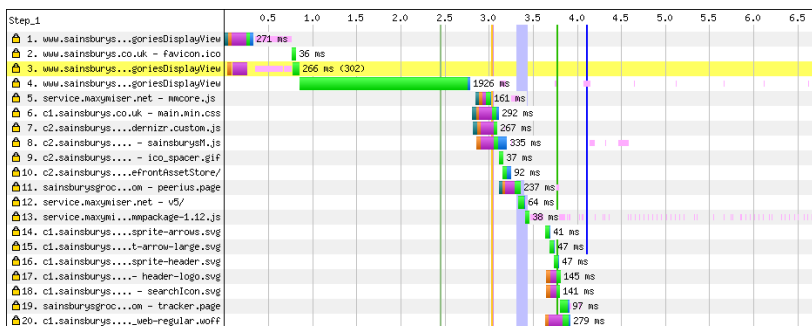
Tools such as WebPagetest [4], Sitespeed.io [5] and their commercial equivalents build on top of the data developer tools generate, and execute tests in a consistent environment, so that we can be confident we can compare test results from day-to-day and week-to-week.

WebPagetest has so many features it's often referred to as the Swiss Army knife of web performance tools.

At the simplest level it enables us to create page load waterfalls, on multiple browsers (including mobile), at different connection speeds, from various locations across the world.

After running a test the first thing we see is a waterfall showing the timings of each request on the page, and markers for browser events such as DOMContentLoaded, and onLoad.

The waterfall allows us to get a picture of how fast or slow the page is (longer bars are bad), how many requests it makes and what types of content they are.

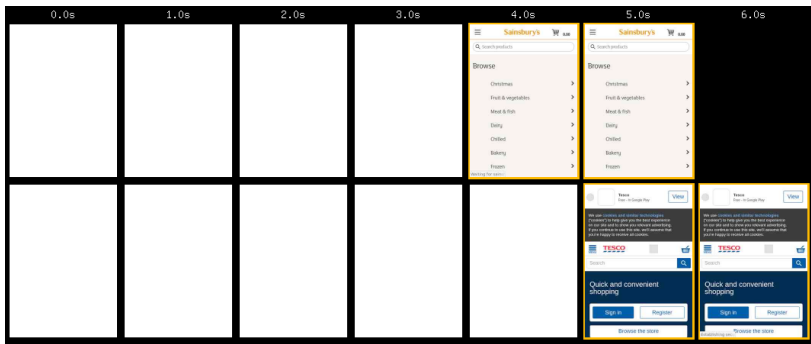


Waterfall showing some of the resources loaded by supermarket Sainsbury's' home page

Interpreting waterfalls can take time to learn and it's not always obvious how the order, in which resources load, affects the visitors' experience.

So one of my favourite features is the filmstrip view and particularly the ability to compare filmstrips from different tests.

We can use them to demonstrate how our site performs compared



Comparison of Sainsbury's' and Tesco's homepages on a mobile device

WebPagetest has an API, so you can run tests regularly and track performance over time, or include testing in a build pipeline so performance regressions are detected early.

Once you start using the API heavily, you'll run into the constraints of the public site being shared with other people and your tests will get stuck behind others. At this point you'll want to consider creating your own private installation either on local hardware, or — if you want an easy option — there are pre-configured images available for AWS.

If you don't fancy running your own instance of WebPagetest or Sitespeed.io, there are similar commercial products such as CalibreApp [6], Performance Analyser [7], and SpeedCurve [8].

It's impossible to give a full overview of all of WebPagetest's features in a short article, so if you want to get a better understanding of the many things it can do, Rick Viscomi, Marcel Duran, and I wrote a book — *Using WebPagetest* [9] — just for you!

Real user monitoring (RUM)

While tools like WebPagetest are great for benchmarking the performance of a set of pages under laboratory conditions, they don't really represent the experience of our real-world visitors.

Performance in the real-world is very diverse, with different visitors having different experiences depending on the device they're using, where they are in the world, the quality of their network connection and even whether they're stood next to the office microwave.

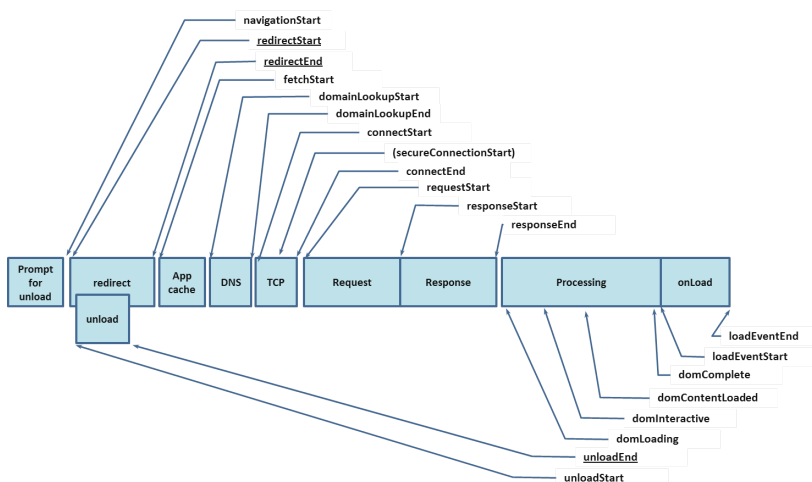
Different types of pages across a site often perform differently, some may be slower to generate, others have more stylesheets, scripts or images, and the large number of pages on most sites makes it impossible to measure them all with tools like WebPagetest.

Fortunately, modern browsers have APIs that provide measurements on the performance of the page and resources on it. We can extract these measurements, beacon them back to a server, enrich them with extra information like location and then analyse the data to build a picture of our visitor's experience and how it affects their behaviour.

There are three well supported APIs that can provide us with measurements on our visitors' experience:

Navigation Timing

Contains the network timing points of the base HTML page, e.g. DNS, connection, server response times, along with details of when events like DOMContentLoaded and onLoad are fired, and how long their handlers take to execute. [10]



Timings provided by the W3C Navigation Timing API

To get an idea of what these timing points look like in practice, you can type `performance.timing` into your browser's DevTools console.

One gotcha to watch out for with Navigation Timing Level 1 is that it uses the number of milliseconds that have passed since 1 January 1970 for its timings, whereas other timing APIs use the number of milli- and microseconds that have passed since Navigation Start. Level 2 of the standard switched to Navigation Start as an epoch, too, but in the meantime some maths is required to align Level 1 timings with the other APIs.

Resource Timing

Captures the network timings for resources (images, scripts, stylesheets etc) included in the page. [11]

Due to privacy and security concerns when a resource comes from a third-party host, by default only a limited set of timing points are available. (Third parties must opt-in via the Timing-Allow-Origin header if they want to make them all available).

The iframe security model may also block access to timing information on resources included within them.

If you'd like some example code that demonstrates Resource Timing, a few years ago I wrote a bookmarklet that uses it to generate a partial page-load waterfall [12].

User Timing

Navigation and Resource Timing focus on network level events and these tell us little about the visitors' experience.

The marks and measures of User Timing [13] enable us to record when events we care about take place and how long they take — performance.mark timestamps a point in time and performance.measure measures the time between two marks.

These allow us to start measuring events that we believe matter to the visitor, for example using our hero image's onload handler to mark when it was loaded or measure how long a route change takes in a Single Page Application.

WebPagetest can display marks and Chrome DevTools measures, so using standards-based approaches gives all our performance measuring tools access to a consistent set of data.

There are other APIs in development (and are currently only supported by a few browsers) which measure when the page was first painted (*Paint Timing*), how long frames took (*Frame Timing*), and highlight long JavaScript tasks (*Long Tasks*).

Perhaps the most exciting of the proposed new APIs is *Element Timing*, which will allow us to declaratively create timing points based on when elements are rendered without needing any JavaScript.

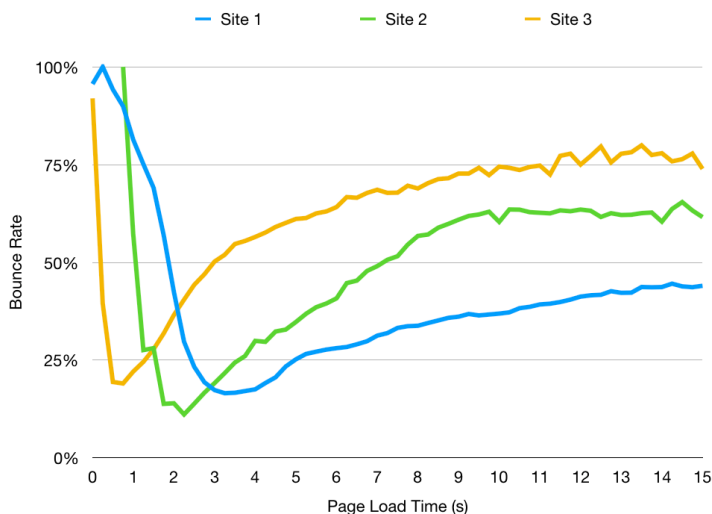
Build your own or buy?

There are libraries on GitHub such as Boomerang [14] and Episodes [15] to help extract this data and beacon it to a server. There are other server libraries to capture the beacons but sanitising, storing and analysing the data is all code you would need to implement if you wanted to build your own RUM solution.

It's not unusual to see beacons with invalid timing data, for example request end occurring before request start, so if you write your own RUM solution, you're going to spend a lot of time sanitising data.

There are many commercial RUM products available, some focus on just the performance measurements while others combine the performance measurements with data on conversions, order values, and engagement to build a picture of how performance affects the success of the site.

Whether you choose to build your own RUM solution or buy a product, I'd encourage you to choose an approach that also captures business metrics such as revenue, so you can combine them and demonstrate the impact of performance on the metrics your business cares about.



How performance affects the bounce rate for three UK sites

Combining the approaches

We've covered three approaches for measuring performance and each has its own strengths, so where do you start and how do they fit together?

I tend to start with real user monitoring and use it to identify pages that are important and should be faster, or visitor populations that are slower and have a lower conversion rate or a higher bounce rate.

Once I've identified the pages, or device experiences I want to analyse, I use either DevTools or WebPagetest to explore further. There's overlap between the two tools, so I tend to use DevTools to interactively explore and examine what's actually happening in the browser, whereas I tend to use WebPagetest to help understand network behaviour, or block third parties to see what impact they have on page performance.

When I want to demonstrate the impact of performance changes, generating a comparison filmstrip in WebPagetest is an ideal way to illustrate the improvement in an easy-to-understand way.

WebPagetest is great when teams want to build performance into their day-to-day work cycle, both via testing key pages within build pipelines, and via hourly or daily tests where the key metrics are charted on dashboards, so performance can be tracked over time.

Faster experiences increase visitor engagement, and measuring performance effectively is key to identifying the visitors we're delivering a poor experience to and the parts of our sites and pages that need the most attention.

Resources

[1] YSlow

<http://yslow.org/>

[2] PageSpeed Insights

<https://developers.google.com/speed/pagespeed/insights/>

[3] Yellow Lab Tools

<http://yellowlab.tools/>

[4] WebPagetest

<https://www.webpagetest.org/>

[5] Sitespeed.io

<https://www.sitespeed.io/>

[6] Calibre

<https://calibreapp.com/>

[7] Performance Analyser

<https://www.nccgroup.trust/uk/our-services/website-performance/web-performance-products/performance-analyser/>

[8] SpeedCurve

<https://speedcurve.com/>

[9] Rick Viscomi, Andy Davies, & Marcel Duran, O'Reilly Books

Using WebPageTest

<https://shop.oreilly.com/product/0636920033592.do>

[10] W3C

Navigation Timing

<https://www.w3.org/TR/navigation-timing/>

[11] W3C

Resource Timing Level 1

<https://www.w3.org/TR/resource-timing/>

[12] Andy Davies

Page Load Waterfalls Bookmarklet

<https://github.com/andydavies/waterfall>

[13] W3C

User Timing

<https://www.w3.org/TR/user-timing/>

[14] Boomerang

<https://github.com/yahoo/boomerang>