

FRONT-END PERFORMANCE



BECAUSE EVERY SECOND COSTS REAL MONEY

Front-end Performance

Copyright © 2017 SitePoint Pty. Ltd.

- Product Manager: Simon Mackie
- English Editor: Ralph Mason
- Technical Editor: Darin Dimitrov
- Cover Designer: Alex Walker

Notice of Rights

All rights reserved. No part of this book may be reproduced, stored in a retrieval system or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embodied in critical articles or reviews.

Notice of Liability

The author and publisher have made every effort to ensure the accuracy of the information herein. However, the information contained in this book is sold without warranty, either express or implied. Neither the authors and SitePoint Pty. Ltd., nor its dealers or distributors will be held liable for any damages to be caused either directly or indirectly by the instructions contained in this book, or by the software or hardware products described herein.

Trademark Notice

Rather than indicating every occurrence of a trademarked name as such, this book uses the names only in an editorial fashion and to the benefit of the trademark owner with no intention of infringement of the trademark.



Published by SitePoint Pty. Ltd.

48 Cambridge Street Collingwood
VIC Australia 3066

Web: www.sitepoint.com
Email: books@sitepoint.com

ISBN: 978-0-9943469-6-4 (print)

ISBN: 978-0-9943470-2-2 (ebook)

Printed and bound in the United States of America

About SitePoint

SitePoint specializes in publishing fun, practical, and easy-to-understand content for web professionals. Visit <http://www.sitepoint.com/> to access our blogs, books, newsletters, articles, and community forums. You'll find a stack of information on JavaScript, PHP, Ruby, mobile development, design, and more.

Table of Contents

Which Browsers Should Your Website Support?

Craig Buckler

Chapter

1

This chapter is part of a series created in partnership with [SiteGround](#). Thank you for supporting the partners who make SitePoint possible.

The question "*which browsers should my website/app support?*" is often raised by clients and developers. The simple answer is a list of the top N mainstream applications. But has that policy become irrelevant?

What are the Most-used Browsers?

The top ten desktop browsers according to [StatCounter](#) for May 2017 were:

- 1 Chrome --- 59.37% market share
- 2 Firefox --- 12.76%
- 3 Safari --- 10.55%
- 4 IE --- 8.32%
- 5 Edge --- 3.42%
- 6 Opera --- 1.99%
- 7 Android (tablet) --- 1.24%
- 8 Yandex Browser --- 0.48%
- 9 UC Browser --- 0.41%
- 10 Coc Coc --- 0.33%

Mobile now accounts for 54.25% of all web use so we also need to examine the top ten phone browsers:

- 1 Chrome --- 49.23%
- 2 Safari --- 17.73%

- 3 UC Browser --- 15.89%
- 4 Samsung Internet --- 6.58%
- 5 Opera --- 5.03%
- 6 Android --- 3.75%
- 7 IEMobile --- 0.68%
- 8 BlackBerry --- 0.26%
- 9 Edge --- 0.15%
- 10 Nokia --- 0.12%

The worldwide statistics don't tell the whole story:

- Patterns vary significantly across regions. For example, Yandex is the second most-used Russian browser (12.7% share). Sogou is the third most-used browser in China (6.5%). Opera Mobile/Mini has a 28% share in Africa.
- New browser releases appear regularly. Chrome, Firefox and Opera receive updates every six weeks; it would be impractical to check versions going back more than a few months.
- The same browsers can work differently across devices and operating systems. Chrome is available for various editions of Windows, macOS, Linux, Android, iOS and ChromeOS, but it's not the same application everywhere.
- There is an exceedingly long tail of old and new, weird and wonderful browsers on a range of devices including games consoles, ebook readers and smart TVs.
- Your site's analytics will never match global statistics.

Are Browsers So Different?

Despite the organic variety of applications, all browsers have the same goal: *to render web pages*. They achieve this with a rendering engine and there is some cross-pollination:

- 1 Webkit is used in Safari on macOS and iOS.
- 2 Blink is a fork of Webkit now used in Chrome, Opera, Vivaldi and Brave.
- 3 Gecko is used in Firefox.
- 4 Trident is used in Internet Explorer.
- 5 EdgeHTML is an update of Trident used in Edge.

The majority of browsers use one of these engines. They're different projects with diverse teams but the companies (mostly) collaborate via the W3C to ensure new technologies are adopted by everyone in the same way. Browsers are closer than they've ever been, and modern smartphone applications are a match for their desktop counterparts. However, no two browsers render in quite the same way. The majority of differences are subtle, but they become more pronounced as you move toward cutting-edge technologies. A particular feature may be fully implemented in one browser, partially implemented in another, and non-existent elsewhere.

Can My Site Work in Every Browser?

Yes. Techniques such as progressive enhancement (PE) establish a baseline (perhaps HTML only) then enhance with CSS and JavaScript when support is available. Recent browsers get a modern layout, animated effects and interactive widgets. Ancient browsers may get unstyled HTML only. Everything else gets something in between. PE works well for content sites and apps with basic form-based functionality. It becomes less practical as you move toward applications with rich custom interfaces. Your new collaborative video editing app is unlikely to work in the decade-old IE7. It may not work on a small screen device over a 3G network. Perhaps it's possible to provide an alternative interface but the result could be a separate, clunky application few would want to use. The cost would be prohibitive given the size of the legacy browser user base.

Site Owner Recommendations

Site owners should appreciate the following fundamentals and constraints of the web. **The web is not print!** Your site/app will not look identical everywhere. Each device has a different OS, browser, screen size, capabilities etc. **Functionality can differ** Your site can work for everyone but experiences and facilities will vary. Even something as basic as a date entry field can have a diverse range of possibilities but, ideally, the core application will remain operable. **Assess your project** Be realistic. Is this a content site, a simple app, a desktop-like application, a fast-action game etc. Establish a base level of browser compatibility. For example, it must work on most two-year-old browsers with a screen width of 600 pixels over a fast Wi-Fi connection. **Assess your audience** Don't rely on global browser statistics. Who are the primary users? Are they IT novices or highly technical? Is it individuals, small companies or government organisations? Do they sit at a desk or are they on the move? No application applies to everyone --- concentrate on the core users first. Examine the analytics of your existing system where possible but appreciate the underlying data. If your app doesn't work in Opera Mini, you're unlikely to have Opera Mini users. Have you blocked a significant proportion of your market? **Change happens** It's amazing that a web page coded twenty years ago works today. It won't necessarily be pretty or usable but browsers remain backward compatible. *(Mostly. The `<blink>` tag can stay dead!)* However, technology evolves. The more complex your site or application, the more likely it will require ongoing maintenance.

Web Developer Recommendations

With a little care it's possible to support a huge variety of browsers. **Embrace the web!** The web is a device-agnostic platform. Content and simpler interfaces can work everywhere: a modern laptop, a feature phone, a games console, IE6, etc. Learn the basics of progressive enhancement. Even if you choose not to adopt it for your full application, there will be pockets of functionality where it becomes invaluable. **Adopt Defensive Development Techniques** Consider the problem before reaching for the nearest pre-written module, library or framework. Understand the consequences of that technology before you start. Frameworks

should provide a browser support list because they have been tested in limited number of applications. Learn about browser limits and quirks. For example, if you're considering an SVG chart, be aware that it can look odd in IE9 to 11 and fail in IE8 and below. That doesn't mean it's a binary choice of rejecting SVGs or abandoning IE support. There are always compromises which do not incur significant development. For example:

- accept SVG rendering is weird but it remains usable
- only show a table of data in IE, or
- provide an SVG download which IE users can open elsewhere.

Test early and test often You cannot possibly test every device, but developing for a single browser is futile. Continually test your project in a variety of applications. Leaving testing to the end will have catastrophic consequences. It's easy for us to blame tools and browser inadequacies, but the majority of issues can be rectified during the development process if they're spotted early. That's not to say everything must work identically in every browser every time. Feature regressions are inevitable. For example:

- Progressive Web Apps do not work offline on iPhones and iPads --- but online operation is fine.
- CSS Grid is not supported in IE --- but float, flexbox or full-width block fallbacks should be acceptable.
- The desktop edition of Firefox does not show a calendar for date fields --- but users can still enter one.

Install a selection of browsers on your development PC. Mac and Linux users can obtain Microsoft Edge and IE testing tools at developer.microsoft.com/microsoft-edge/. It's more difficult for Windows and Linux users to test Safari; online test services such as [BrowserStack](https://www.browserstack.com/) are the easiest option. Modern browsers have excellent mobile emulation facilities, but use a few real devices to appreciate touch control and performance on slower hardware and networks.

Use HTTPS on your end The web is gradually making HTTPS the preferred protocol, and this trend is going to continue. Google Chrome is even starting to

indicate that non-HTTPS sites as insecure, which is a good reason for you to configure your site to use HTTPS. Our web hosting partner, [SiteGround](#), for example, made it easy for their clients to make the move to HTTPS. To do that, they automated the installation of Let's Encrypt SSL certificates for all new WordPress accounts, and for existing ones, they made the switch to HTTPS possible with just a click.

You Haven't Answered the Question!

The question "*which browsers should you support?*" has become too restrictive. Presume your answer was just "Chrome":

- *which devices and OS is it running on?*
- *what range of screen sizes will be supported?*
- *which version are you referring to? The latest? Chrome 10 and above?*
- *what happens when a new version of Chrome is released?*
- *what will happen in other browsers when Chrome effectively becomes your application's runtime?*

Providing a browser support list has become impractical for client-facing projects. Perhaps the best answer is: "*we'll develop your project according to presumed demographics then test as in many devices, OSes, browsers, and versions as possible according to budget and time constraints*". Even then, you'll miss that aging Blackberry the CEO insists on using. Develop for the web --- *not* browsers.

Five Techniques to Lazy Load Images for Website Performance

Maria Antonietta Perna

Chapter

2

This article is part of a series created in partnership with [SiteGround](#). Thank you for supporting the partners who make SitePoint possible.

With images making up a whopping 65% of all web content, page load time on websites can easily become an issue.

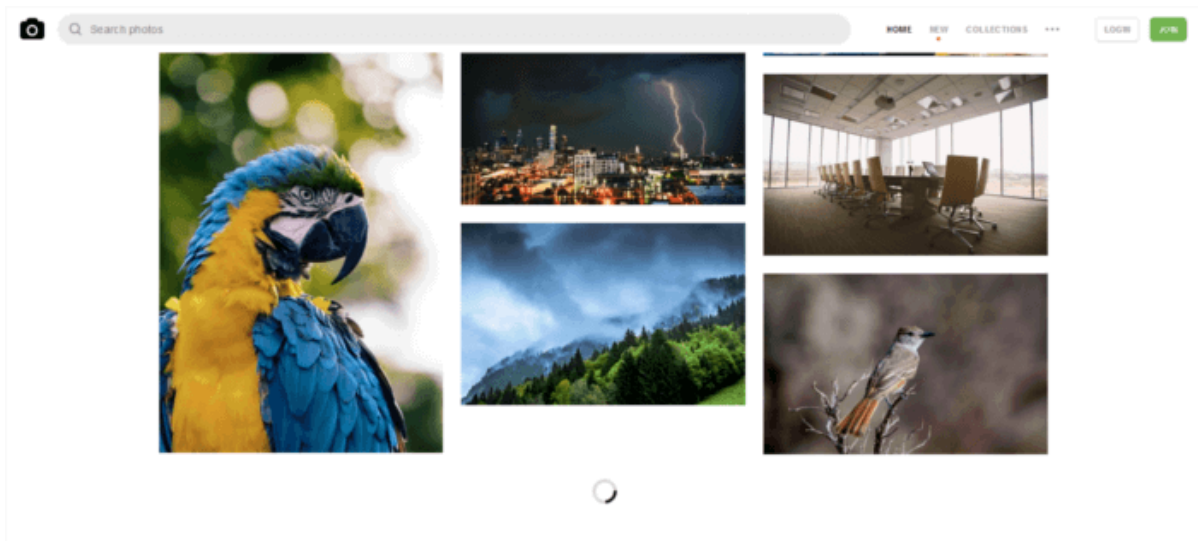
Even when properly optimized, images can weigh quite a bit. This can have a negative impact on the time visitors have to wait before they can access content on your website. Chances are, they get impatient and navigate somewhere else, unless you come up with a solution to image loading that doesn't interfere with the perception of speed.

In this article, you will learn about five approaches to lazy loading images that you can add to your web optimization toolkit to improve the user experience on your website.

What Is Lazy Loading?

Lazy loading images means loading images on websites asynchronously --- that is, after the above-the-fold content is fully loaded, or even conditionally, only when they appear in the browser's viewport. This means that if users don't scroll all the way down, images placed at the bottom of the page won't even be loaded.

A number of websites use this approach, but it's especially noticeable on image-heavy sites. Try browsing your favorite online hunting ground for high-res photos, and you'll soon realize how the website loads just a limited number of images. As you scroll down the page, you'll see placeholder images quickly filling up with real images for preview. For instance, notice the loader on [Unsplash.com](#): scrolling that portion of the page into view triggers the replacement of a placeholder with a full-res photo:



Why Should You Care About Lazy Loading Images?

There are at least a couple of excellent reasons why you should consider lazy loading images for your website:

- If your website uses JavaScript to display content or provide some kind of functionality to users, loading the DOM quickly becomes critical. It's common for scripts to wait until the DOM has completely loaded before they start running. On a site with a significant number of images, lazy loading --- or loading images asynchronously --- could make the difference between users staying or leaving your website.
- Since most lazy loading solutions work by loading images only if the user has scrolled to the location where images would be visible inside the viewport, those images will never be loaded if users never get to that point. This means considerable savings in bandwidth, for which most users, especially those accessing the web on mobile devices and slow-connections, will be thanking you.

Well, lazy loading images helps with website performance, but what's the best way to go about it?

There is no perfect way.

If you live and breathe JavaScript, implementing your own lazy loading solution shouldn't be an issue. Nothing gives you more control than coding something yourself.

Alternatively, you can browse the web for viable approaches and start experimenting with them. I did just that and came across these five interesting techniques.

#1 David Walsh's Simple Image Lazy Load and Fade

David Walsh has proposed his own custom script for lazy loading images. Here's a simplified version:

The `src` attribute of the `img` tag is replaced with a `data-src` attribute in the markup:

```

```

In the CSS, `img` elements with a `data-src` attribute are hidden. Once loaded, images will appear with a nice fade-in effect using CSS transitions:

```
img { opacity: 1; transition: opacity 0.3s; }  
img[data-src] { opacity: 0;  
}
```

JavaScript then adds the `src` attribute to each `img` element and gives it the

value of their respective `data-src` attributes. Once images have finished loading, the script removes the `data-src` attribute from `img` elements altogether:

```
.forEach.call(document.querySelectorAll('img[data-src]'),
function(img) {
    img.setAttribute('src',
    img.getAttribute('data-src'));
    img.onload = function() {
    img.removeAttribute('data-src');
    };
});
```

David Walsh also offers a fallback solution to cover cases where JavaScript fails, which you can find out more about on [his blog](#).

The merit of this solution: it's a breeze to implement and it's effective.

On the flip side, this method doesn't include loading on scroll functionality. In other words, all images are loaded by the browser, whether users have scrolled them into view or not. Therefore, you get the advantage of a fast loading page because images are loaded after the HTML content. However, you don't get the saving on bandwidth that comes with preventing unnecessary image data from being loaded when visitors don't view the entire page content.

#2 Robin Osborne's Progressively Enhanced Lazy Loading

Robin Osborne suggests a super ingenious solution based on [progressive enhancement](#). In this case, lazy loading itself, which is achieved using JavaScript, is considered the enhancement over regular HTML and CSS.

Why progressive enhancement? Well, if you display images using a JavaScript-based solution, what happens if JavaScript is disabled or an error occurs which prevents the script from working as expected? In this case, without progressive enhancement, users are likely to see no images at all. Not cool.

You can see the details of a basic version of Osborne's solution in this [Pen](#), and a more comprehensive one, which takes into account the case for *broken JavaScript*, in this other [Pen here](#).

This technique has a number of advantages:

- The progressive enhancement approach ensures users always have access to content.
- Not only does it cater for a situation where JavaScript is not available, but also for those cases when JavaScript is *broken*: we all know how error-prone scripts can be, especially in an environment where a significant number of scripts are running.
- It lazy loads images on scroll, so not all images will be loaded if users don't scroll to their location in the browser.
- It doesn't rely on any external dependencies, hence no frameworks or plugins are necessary.

You can learn all the details of Robin Osborne's approach on [his blog](#).

#3 Lazy Load XT jQuery Plugin

A quick and easy alternative for implementing lazy loading of images is to let a JavaScript/jQuery plugin do the heavy lifting for you.

Lazy Load XT is a feature-packed jQuery plugin. You can opt for a simplified version called `jquery.lazyloadxt.js`, which lets you just lazy load images. Alternatively, you can use `jquery.lazyloadxt.extra.js`, which is an extended version of the plugin. With the extended version, you can lazy load iframes, videos, and generally all tags that use a `src` attribute.

To include Lazy Load XT in your project, at the bottom of your HTML page before the closing `</body>` tag, add the jQuery library, followed by one of the two Lazy Load XT flavors mentioned above. For instance:

```
<script src="jquery.js"></script>
<script src="jquery.lazyloadxt.js"></script>
```

If you don't want to use jQuery, Lazy Load XT offers a much lighter option, a small script called `jqlight.lazyloadxt.min.js`:

```
<script src="jqlight.lazyloadxt.js"></script>
<script src="jquery.lazyloadxt.js"></script>
```

In your HTML document, mark up images using a `data-src` attribute instead of the regular `src` attribute, like this:

```

```

You can then leave the plugin to initialize itself, or you can manually initialize it yourself. For instance, to initialize a selection of elements write:

```
$(elements).lazyLoadXT();
```

This plugin makes tons of add-ons for extra functionality available. To mention just a couple:

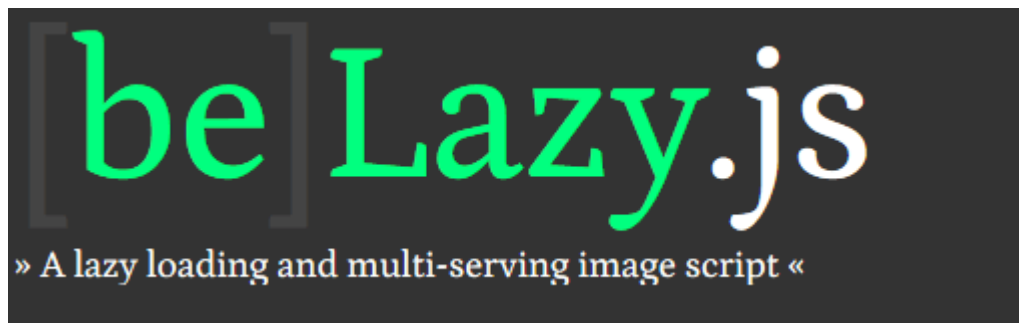
- By adding `jquery.lazyloadxt.spinner.css`, you can display an animated spinner as the image is loading.
- You can add all the [Animate.css](#) fun effects for image loading if you just include `animate.min.css` in your project and write this line of code in your JavaScript file: `$.lazyLoadXT.onload.addClass = 'animated bounceOutLeft'`; . Of course, you can replace `bounceOutLeft` with any of the effects `Animate.css` provides.

Among the advantages of Lazy Load XT and its add-ons are:

- CDN hosting support so you don't need to download Lazy Load XT scripts to your server.
- Wide browser support, including IE6-11 and Opera Mini.
- You can lazy load images on the page, in scrollable containers, in both vertical and horizontal scroll layouts, and in infinite scrolling scenarios.
- Using add-ons you can create great transition effects, implement support for retina screens, lazy load background images, and much more.
- You can lazy load different media types.
- The documentation shows how you can counteract the eventuality of browsers with JavaScript disabled.
- You don't need to include the full jQuery library to use this plugin for lazy loading images.

For a full list of options and add-ons, visit the [Lazy Load XT repo on GitHub](#).

#4 bLazy.js — Vanilla JavaScript Plugin



bLazy is a smart vanilla JavaScript plugin for lazy loading images. More specifically:

bLazy is a lightweight lazy loading image script (less than 1.2KB minified and gzipped). It lets you lazy load and multi-serve your images so you can save bandwidth and server requests. The user will have faster load times and save data loaded if he/she doesn't browse the whole page.

bLazy website

This small, dependency-free plugin lets you:

- lazy load both embedded images and background images
- serve different images according to screen size and high resolution screens
- lazy load everything with a `src` attribute, e.g., iframes, HTML5 videos, scripts, unity games etc.
- lazy load images inside a scrolling container
- support legacy browsers, back to IE7 and IE8
- use a CDN, so you don't need bLazy hosted on your server.

Here's a basic implementation.

The markup:

```
<img class="b-lazy" src="placeholder.gif" data-src="image.jpg" alt="te
```

You need to modify the regular `img` tag as follows:

- Add a class of `.b-lazy`.
- Use a placeholder image as value for the `src` attribute. To save HTTP requests, you can also use an inline base64-encoded transparent gif. But beware, doing so won't have the benefits of caching on subsequent pages

where you use the same image.

- The `data-src` attribute points to the image you want to lazy load.

The JavaScript: enter a simple call to `bLazy` and fine-tune with a map of options:

```
var bLazy = new Blazy({  
  //options here  
});
```

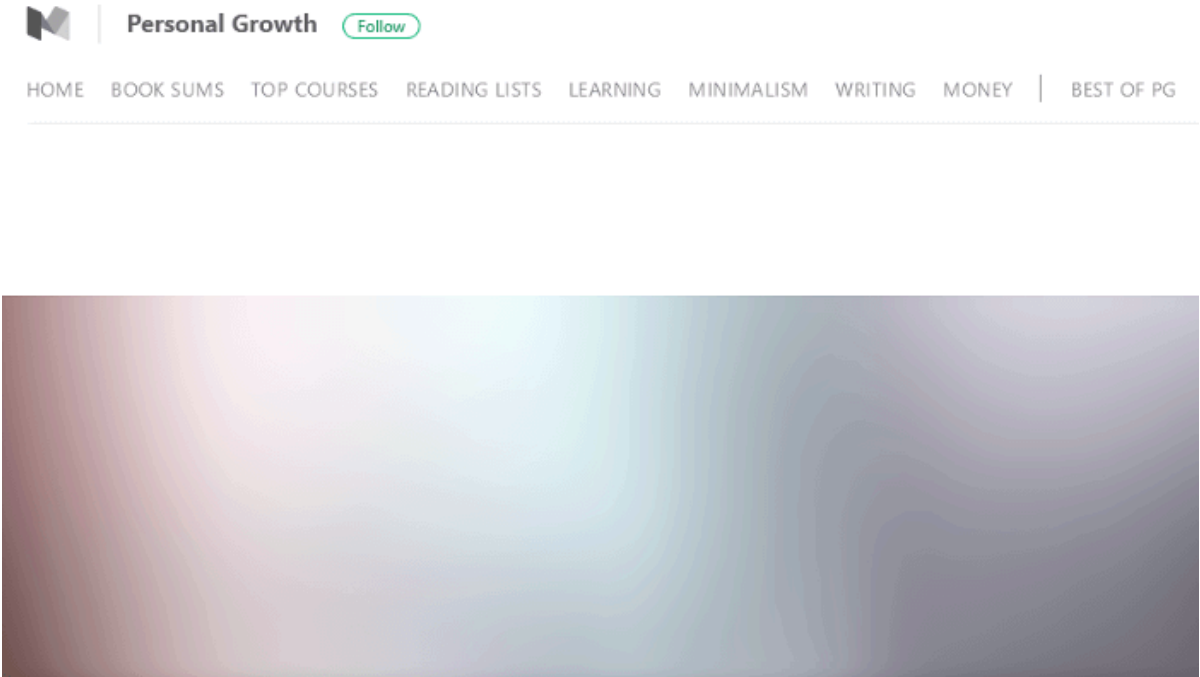
To learn more on `bLazy` and its available options, follow these links:

- [bLazy.js – A lazyload image script](#)
- [bLazy.js Website](#)
- [bLazy.js Examples](#)
- [bLazy.js Project on GitHub](#).

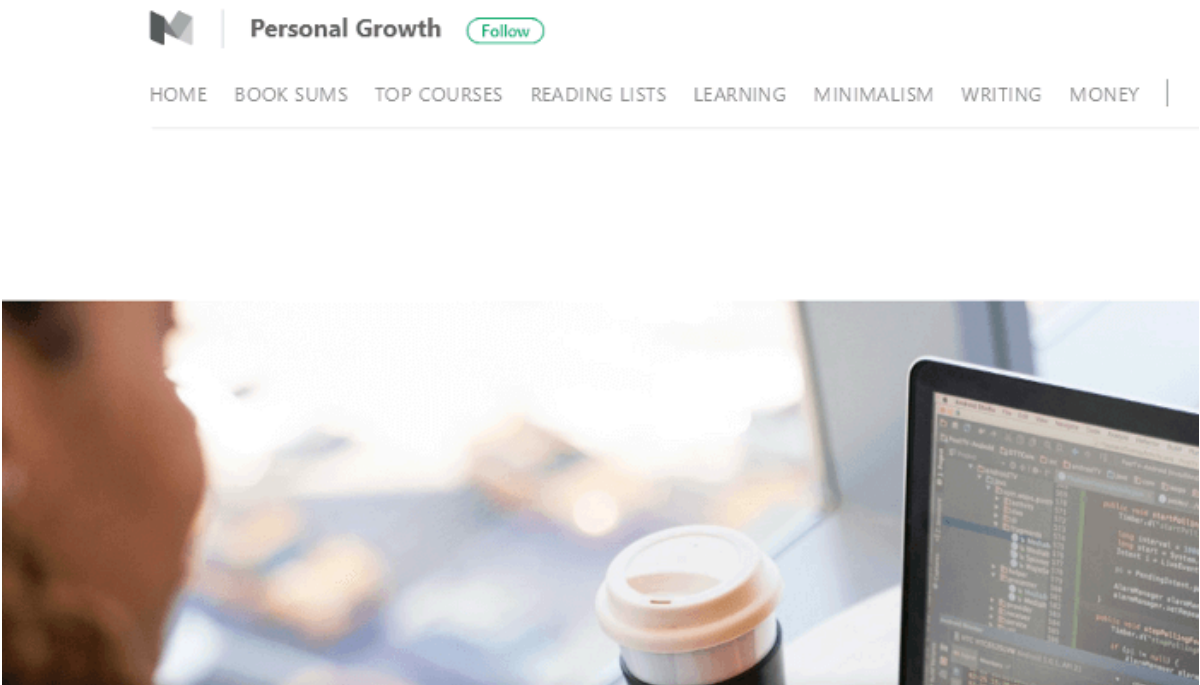
#5 Lazy Loading with Blurred Image Effect

If you are a [Medium](#) reader, you have certainly noticed how the site loads the main image inside a post.

The first thing you see is a blurred, low-resolution copy of the image, while its high-res version is being lazy loaded:



Blurred placeholder image on Medium website.



High-res, lazy loaded image on Medium website.

You can lazy load images with this interesting blurring effect in a number of ways.

My favorite technique is by Craig Buckler. Here's all the goodness of this solution:

- Performance: only 463 bytes of CSS and 1,007 bytes of minified JavaScript code
- Support for retina screens
- Dependency-free: no jQuery or other libraries and frameworks required
- Progressively enhanced to counteract older browsers and failing JavaScript

You can read all about it in [How to Build Your Own Progressive Image Loader](#) and download the code on the project's [GitHub repo](#).

Conclusion

And there you have it --- five ways of lazy loading images you can start to experiment with and test out in your projects.

Optimizing CSS: ID Selectors and Other Myths

Ivan Čurić

Chapter

3

In today's typical scenario, where the average website ships 500KB of gzipped JavaScript and 1.5MB of images, running on a midrange Android device via 3G with a 400ms round trip time, CSS selector performance is the least of our problems.

Still, there's something to be said about the topic, especially to weed out some of the myths and legends surrounding them. So let's dive right in.

The Basics of CSS Parsing

First, to get on the same page --- this article isn't about the performance of CSS properties and values. What we're covering here is the performance cost of the selectors themselves. I'll be focusing on the Blink rendering engine, specifically Chrome 62.

The selectors can be split into a few groups and (roughly) sorted from the least to most expensive:

rank	type	example
1.	ID	<code>#classID</code>
2.	Class	<code>.class</code>
3.	Tag	<code>div</code>
4.	General and adjacent sibling	<code>div ~ a, div + a</code>
5.	Child and descendant	<code>div > a, div a</code>
6.	Universal	<code>*</code>
7.	Attribute	<code>[type="text"]</code>
8.	Pseudo-classes and elements	<code>a:first-of-type, a:hover</code>

Does this mean that you should only use IDs and classes? Well ... not really. It depends. First, let's cover how browsers interpret CSS selectors.

Browsers read CSS from right to left. The rightmost selector in a compound selector is known as the **key selector**. So, for instance, in `#id .class > ul a`, the key selector is `a`. The browser first matches all key selectors. In this case, it finds all elements on the page that match the `a` selector. It then finds all `ul` elements on the page and filters the `as` down to just those elements that are descendants of `uls` --- and so on until it reaches the leftmost selector.

Therefore, the shorter the selector, the better. If possible, make sure that the key selector is a class or an ID to keep it fast and specific.

Measuring the Performance

Ben Frain created [a series of tests](#) to measure selector performance back in 2014. The test consisted of an enormous DOM comprising 1000 identical elements, and measuring the speed it took to parse various selectors, ranging from IDs to some seriously complicated and long compound selectors. What he found was that the delta between the slowest and fastest selector was ~15ms.

However, that was back in 2014. Things have changed a lot since then, and memorizing rules is all but useless in the ever-changing browser landscape. Always remember to do your own tests, especially when performance is concerned.

I went to do my own tests, and for that I used Paul Lewis' test mentioned in [Paul Irish's comment](#) expressing concern over the useful, yet convoluted "[quantity selectors](#)":

These selectors are among the slowest possible. ~500 slower than something wild like `"div.box:not(:empty):last-of-type .title"`. Test page <http://jsbin.com/gozula/1/quiet>

The test was bumped up a bit, to 50000 elements, and you can [test it out yourself](#). I did an average of 10 runs on my 2014 MacBook Pro, and what I got was the following:

Selector	Query Time (ms)
<code>div</code>	4.8740
<code>.box</code>	3.625
<code>.box > .title</code>	4.4587
<code>.box .title</code>	4.5161
<code>.box ~ .box</code>	4.7082
<code>.box + .box</code>	4.6611
<code>.box:last-of-type</code>	3.944
<code>.box:nth-of-type(2n - 1)</code>	16.8491
<code>.box:not(:last-of-type)</code>	5.8947
<code>.box:not(:empty):last-of-type .title</code>	8.0202
<code>.box:nth-last-child(n+6) ~ div</code>	20.8710

The results will of course vary depending on whether you use `querySelector` or `querySelectorAll`, and the number of matching nodes on the page, but `querySelectorAll` comes closer to the real use case of CSS, which is targeting all matching elements.

Even in such an extreme case, with 50000 elements to match, and using some really insane selectors like the last one, we find that the slowest one is ~20ms, while the fastest is the simple class at ~3.5ms. Not really that much of a difference. In a realistic, more "tame" DOM, with around 1000–5000 nodes, you can expect those results to drop by a factor of 10, bringing them to sub-millisecond parsing speeds.

What we can see from this test is that it's not really worth it to worry over CSS selector performance. Just don't overdo it with pseudo selectors and really long selectors. We can also see how Blink improved in the last two years. Instead of the stated ~500x slowdown for a "quantity selector"

`(.box:nth-last-child(n+6) ~ div)` compared to an "insanity selector" `(.box:not(:empty):last-of-type .title)`, we only see a ~1.5x slowdown. That's an amazing improvement, and we can only expect browsers to get better, making CSS selector performance even less impactful.

You *should*, however, stick to using classes whenever possible, and adopt some sort of namespacing convention like BEM, SMACSS or OOCSS, since it will not only help your website's performance but vastly help with code maintainability. Overqualified compound selectors, especially when used with tag and universal selectors --- such as `.header nav ul > li a > .inner` --- are extremely brittle and a source of many unforeseen errors. They are also a nightmare to maintain, especially if you inherit the code from someone else.

Quality over Quantity

A bigger problem of simply having expensive selectors is having *a lot* of them. This is known as "style bloat", and you've probably seen the problem a lot. Typical examples are sites which import entire CSS frameworks like Bootstrap or Foundation, while using less than 10% of the transferred CSS. Another example is seen in old, never-refactored projects whose CSS has devolved into, as I like to call them, "Chronological Style Sheets" --- CSS with a ton of appended classes to the end of the file as the project has changed and grown, now looking more like an overgrown garden full of weeds.

Not only does a large CSS file take longer to transfer, (and network is the *biggest* bottleneck in website performance), they also take longer to parse. As well as constructing the DOM from your HTML, the browser needs to construct a CSSOM (CSS Object Model) to compare it with the DOM and match the selectors.

So, keep your styles lean and DRY, don't include everything and the kitchen sink, load what you need and when you need it, and use UNCSS if you need to.

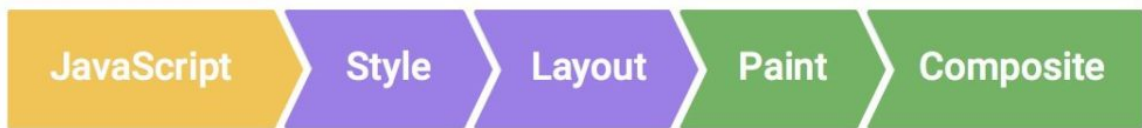
If you want to dig more into how the browsers parse CSS, [check out Nicole](#)

[Sullivan's post on Webkit](#), [Ilya Grigorik's article on how Blink does it](#), or [Lin Clark's article on Mozilla's new Stylo CSS engine](#).

The Elephant in the Room: Style Invalidation

What we've covered so far is fine, but we've only discussed a single rendering pass. Today's websites are no longer static documents, but resemble apps with dynamic content users can interact with.

This complicates things, since parsing CSS is only a single step in the browser rendering pipeline. Here's a render-oriented view of how a browser renders a single frame to the screen (source: [Google](#)):



We won't be going into JavaScript performance and compositing, but will focus instead on the purple part --- style parsing and laying out the elements.

After constructing the DOM and CSSOM, the browser needs to combine the two into a render tree before finally painting it on the screen. In that step, the browser needs to figure out the computed CSS for each matching element. You can see this yourself in the **Elements > Styles** panel of the developer tools. It takes all the matching styles, the cascade, and browser-specific user agent styles to construct the final, computed CSS for the element.

It can then proceed to the layout (also known as reflow) step, where it computes the geometry, and constructs the box model of the page, placing each element on its respective position on the viewport. Layout is the most computationally intensive part of this process.

Finally, the browser converts each node in the render tree to actual pixels on the

screen in the paint stage.

Now, what happens when we mutate the DOM by *changing* some classes on the page, adding or removing some nodes, modifying some attributes, or in any way messing with the HTML (or the stylesheets themselves)?

We invalidate the computed styles and the browser needs to invalidate *everything* down the tree of the matched selectors. While today's browsers are much smarter, it used to be the case that if you changed a class on the `body` element, all the descendant elements needed to have their computed styles recalculated.

One way to avoid this issue is to reduce the complexity of your selectors. Instead of writing `#nav > .list > li > a`, use a single selector, like `.nav-link`. That way, you reduce the scope of style invalidation, since if you modify anything inside the `#nav`, you won't trigger recalculations for the entire node.

Another way is to reduce the scope --- such as the number of invalidated elements. Be specific with your CSS. Keep this in mind especially during animations, where the browser has only ~10ms to do all the work required.

If you want to get down to the nitty gritty details of style invalidation, I recommend reading [Style Invalidation in Blink](#).

Conclusion

To sum it up, you shouldn't worry about selector performance, unless you *really* go overboard. While the topic was all the rage in 2012, browsers have gotten *a lot* faster and smarter since. Even Google doesn't worry about it anymore. If you check out Google's [Page Speed Insights page](#), you won't see the rule "Use efficient CSS selectors" which was removed in 2013.

Instead, focus on making your CSS maintainable and readable. Your colleagues and your future self will thank you for it. Try to optimize the CSS delivery by

including only the used styles. And after that, get to know the rendering pipeline. Unlike selectors, styles themselves *can* be expensive, and the difference between a janky site and a smooth one can often be found in how the CSS is implemented.

And as a final note: always do your own tests.

Don't just believe what someone wrote on the internet a few years ago. The landscape is changing drastically and at an incredible pace. What's relevant today might become obsolete sooner than you know.

Optimizing CSS: Tweaking Animation Performance with DevTools

Maria Antonietta Perna

Chapter

4

This article is part of a series created in partnership with [SiteGround](#). Thank you for supporting the partners who make SitePoint possible.

CSS animations are known to be super performant. Although this is the case for simple animations on a few elements, if you didn't code your animations with performance in mind and add more complexity, website users will soon take notice and possibly get annoyed.

In this article, I introduce some useful browser DevTools features that will enable you to check what happens under the hood when animating with CSS. This way, when an animation looks a bit choppy, you'll have a better idea why and what you can do to fix it.

Developer Tools for CSS Performance

Your animations need to hit 60 fps (frames per second) to run fluidly in the browser. The lower the rate, the worse your animation will look. This means the browser has no more than about 16 milliseconds to do its job for one frame. But what does it do during that time? And how would you know if your browser is keeping up with the desired framerate?

I think nothing beats user experience when it comes to assessing the quality of an animation. However, developer tools in modern browsers, while not always 100% reliable, have been getting smarter and smarter, and there's quite a bit you can do to review, edit and debug your code using them.

This is also true when you need to check framerate and CSS animation performance. Here's how it works.

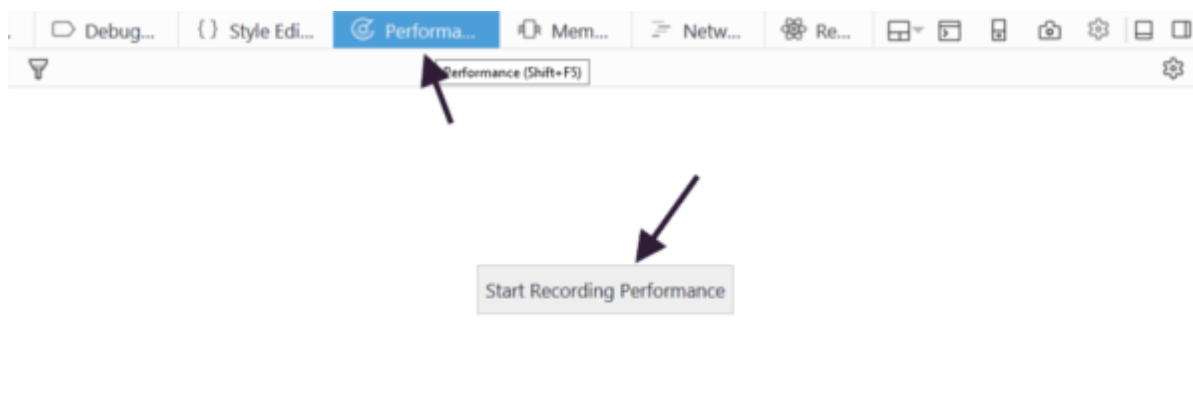
Exploring the Performance Tool in Firefox

In this article I use the Firefox Performance tool. The other big contender is the Chrome Performance Tool. You can pick your favorite, as both browsers offer powerful performance features.

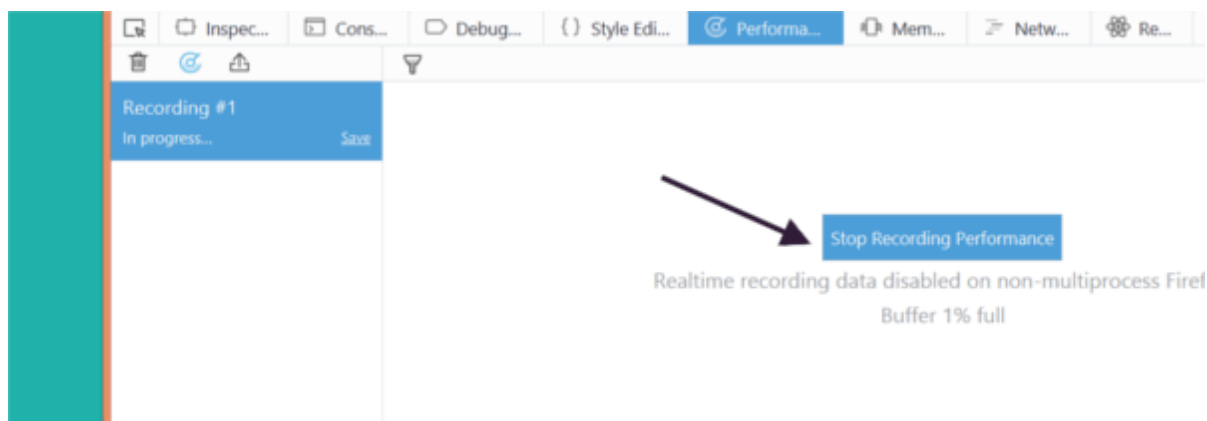
To open the developer tools in Firefox, choose one of these options:

- Right-click on your web page and choose *Inspect Element* in the context menu
- If you use the keyboard, press Ctrl + Shift + I on Windows and Linux or Cmd + Opt + I on macOS.

Next, click on the *Performance* tab. Here, you'll find the button that lets you start a recording of your website's performance:



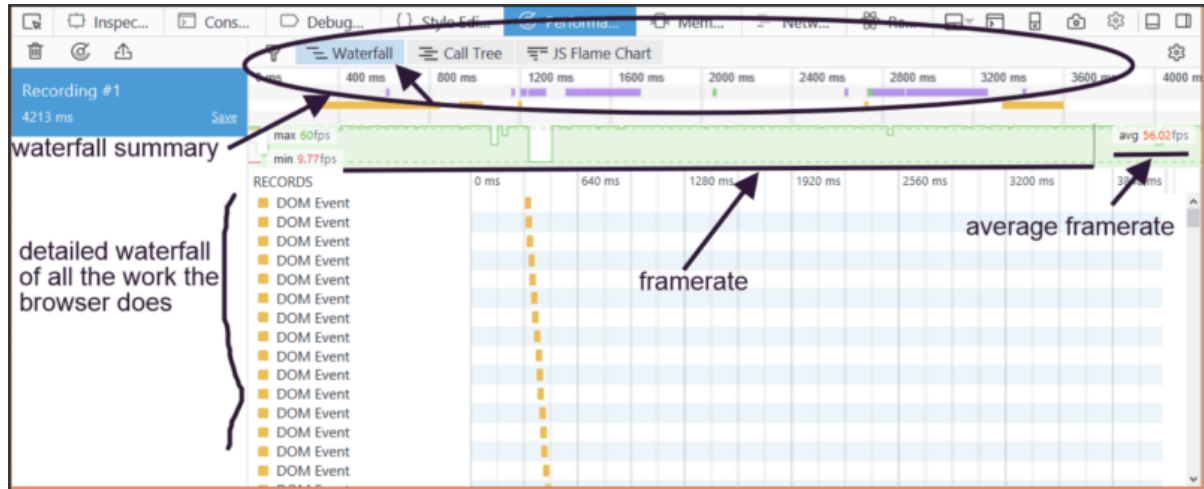
Press that button and wait for a few seconds or perform some action on the page. When you're done, click the *Stop Recording Performance* button:



In a split second, Firefox presents you with tons of well-organized data that will

help you make sense of which issues your code is suffering from.

The result of a recording inside the *Performance* panel looks something like this:

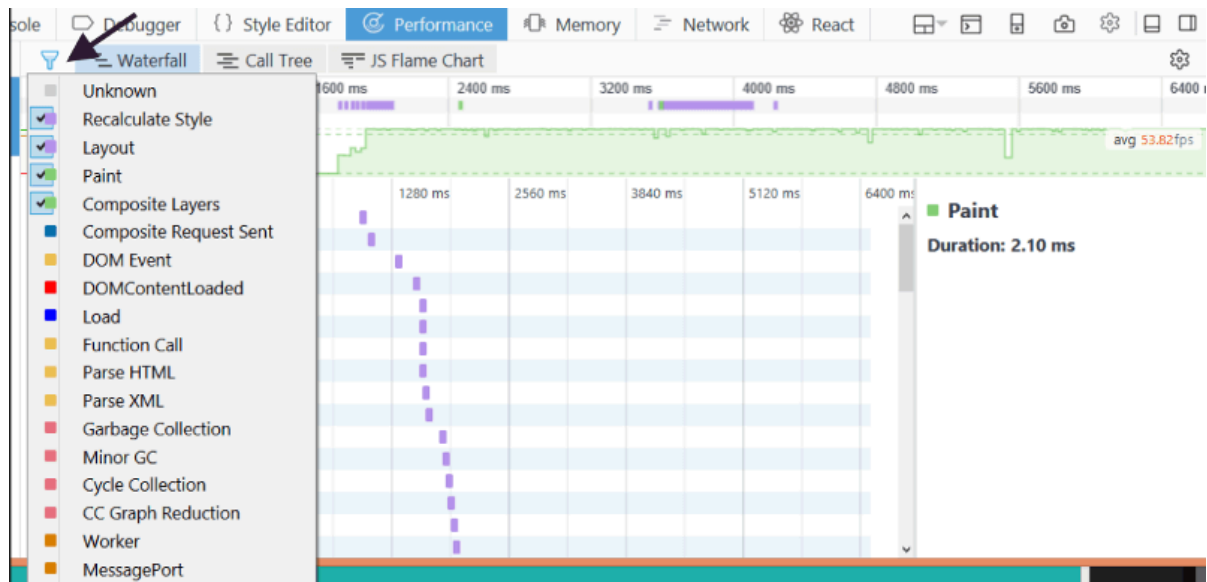


The *Waterfall* section is perfect for checking issues related to CSS transitions and keyframe animations. Other sections are the *Call Tree* and the *JS Flame Chart*, which you can use to find out about bottlenecks in your JavaScript code.

The Waterfall has a summary section at the top and a detailed breakdown. In both, the data is color-coded:

- Yellow bars refer to JavaScript operations.
- Purple bars refer to calculating HTML elements' CSS styles (recalculate styles) and laying out your page (layout). Layout operations are quite expensive for the browser to perform, so if you animate properties that involve repeated layouts (also known as *reflows* --- such as `margin`, `padding`, `top`, `left`, etc. --- the results could be janky.
- Green bars refer to painting your elements into one or more bitmaps (Paint). Animating properties like `color`, `background-color`, `box-shadow`, etc., involves costly paint operations, which could be the cause of sluggish animations and poor user experience.

You can also filter the type of data you want to inspect. For instance, I'm interested only in CSS-related data, so I can deselect everything else by clicking on the filter icon at the top left of the screen:



The big green bar below the Waterfall summary represents information on the framerate.

A healthy representation would look quite high, but most importantly, consistent --- that is, without too many deep gaps.

Let's illustrate this with an example.

The Performance tool in action

This is a simple [CSS animation](#) using the `@keyframes` keyword. The test page looks like this:

CSS Animation Test



CSS Animation Test



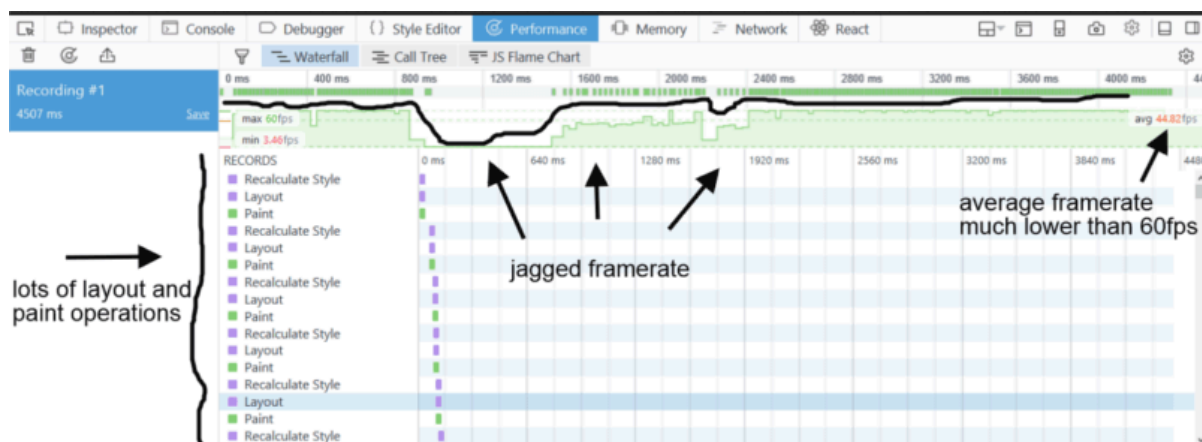
The rectangular purple box slides in and out of view in an infinite cycle.

I've done this by animating the `margin-left` property of the `<div>` element that represents the rectangular box on the screen. Here's what the `@keyframes`

animation block looks like:

```
@keyframes slide-margin {
  100% {
    margin-left: 0;
  }
}
```

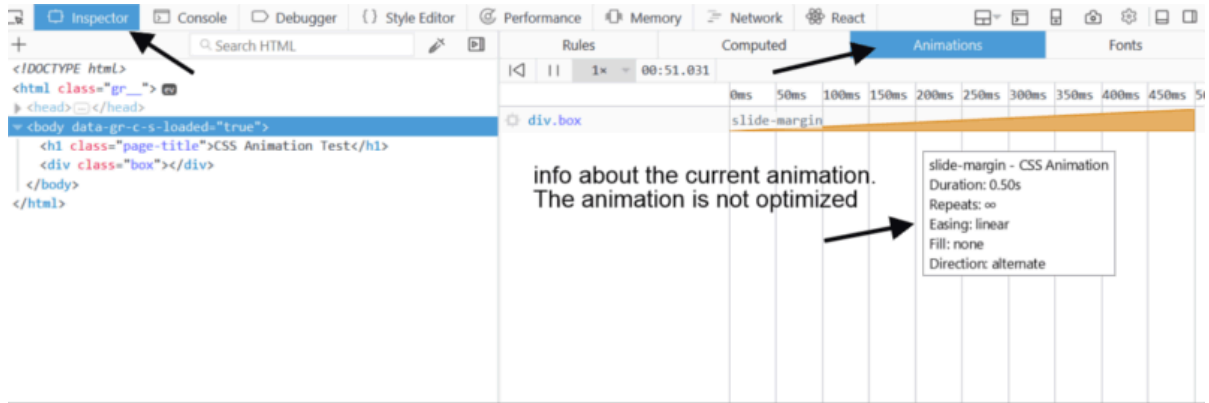
The performance data I get from this animation looks like this:



The framerate visual looks a bit jagged and the framerate is an average of 44.82 fps, which is a bit low.

Also, notice all the layout and paint operations that take place during the animation. These are costly operations the browser performs on its main thread, which has a negative impact on performance.

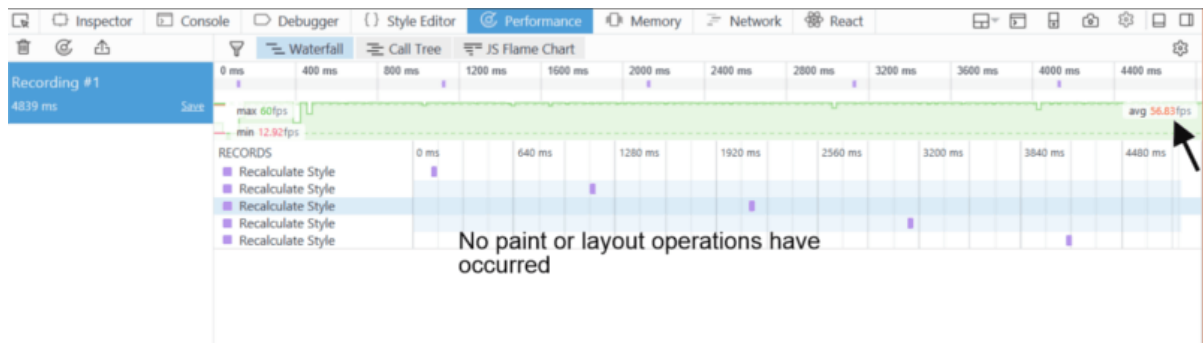
Finally, if you access the *Inspector* tool, click in the *Animation* section and hover over the animation name, an information box pops up with all the relevant data about the current animation. If your animation were optimized, there would be a message stating the fact. In this case, there is no message:



Now, I'm going to change my code and make a new recording as the browser animates the CSS `translate3d()` property using this `@keyframes` block:

```
@keyframes slide-three-d {
  100% { transform: translate3d(0, 0, 0);
}
```

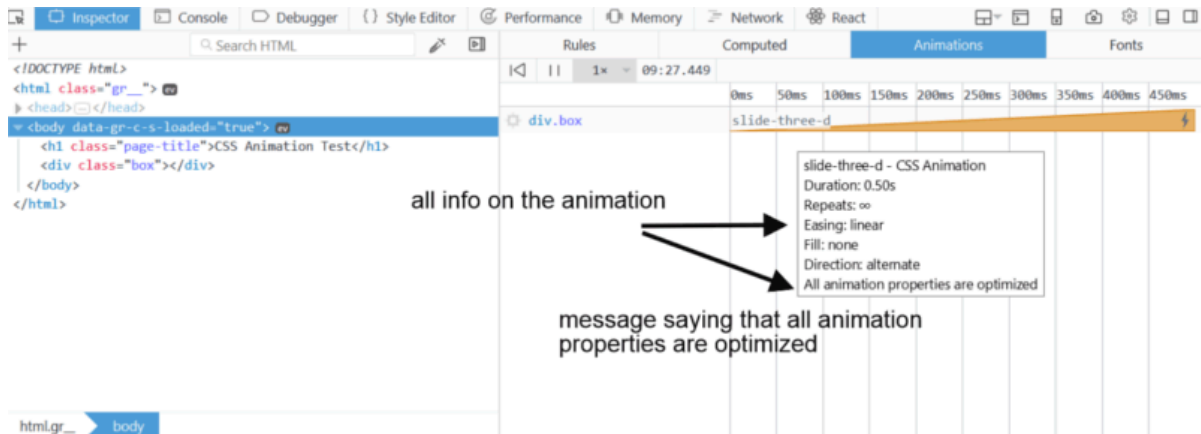
This is what the recording of the performance looks like:



Now the framerate is higher (56.83 fps) and the waterfall is showing no costly layout and paint operations.

Also, if you open the *Inspector* tab of the developer tools, access the *Animation*

panel and hover the mouse over the name of the animation, you can see something like this:



The info box relative to the animation name states that all animations are optimized, which is good news for your website visitors.

Only Animate CSS *Opacity*, *Transforms* and *Filters*

You've probably heard this piece of advice before, but just in case, it's worth going over it again: if you want your animations to run smoothly, animate only CSS opacity, transforms and filters. Animating everything else will put your browser under pressure to perform costly tasks in very little time, which often doesn't yield the best results.

As the Performance tool in your browser confirms, repeated layout and paint operations are not your friend.

However, each browser handles CSS properties a bit differently. If you want to know exactly which browser triggers layout and paint operations for which properties (especially when updating values for those properties, which is what is involved in web animation), head over to [CSS Triggers](#).

To ensure performant animations, a popular approach is to force the browser to

hand over the work of changing some properties to the GPU (Graphical Processing Unit), which relieves the browser's main thread of some pressure and takes advantage of hardware acceleration. You can do so by using the will-change CSS property, or the `translateZ(0)` and `translate3d(0,0,0)` hacks. All of these tricks will work, but if you overdo it you could actually get exactly what you're trying to avoid, i.e., janky animations.

I'm not going into the details of hardware acceleration for web animation performance, but if you'd like to dig deeper, take a look at the resources listed below.

Resources

- [High Performance Animations](#) by Paul Lewis and Paul Irish
- [CSS animations and transitions performance: looking inside the browser](#) by Max Vujovic
- [Animations and Performance](#) by Paul Lewis and Sam Thorogood
- [Stick to Compositor-Only Properties and Manage Layer Count](#) by Paul Lewis
- [Tricks for GPU Composited CSS](#) by Sara Soueidan
- [An Introduction to the CSS will-change Property](#) by Nick Salloum
- [Animating CSS properties](#) by MDN

Lightning Fast Websites with Prefetching

Maria Antonietta Perna

Chapter

5

This article is part of a series created in partnership with [SiteGround](#). Thank you for supporting the partners who make SitePoint possible.

In this article, I'll discuss prefetching, what it is, and how developers can use it to wow visitors with high-performance websites.

What Is Prefetching?

Although optimizing a website for initial page load is great, for the highly interactive websites users expect today, this might not be enough.

What if browsers knew which link users were about to click, or which page they were about to visit next, so that content could automagically appear on their screen at the speed of light?

Well, browsers *can* do that now. In fact, some major browsers are smart enough to make these kinds of guesses based on visitors' browsing patterns, document markup and structure, the user's device, connectivity, etc. Therefore, browsers already try to anticipate the resources needed to build the page visitors are likely to access, get those resources ready and display the page at warp speed when users request it. However, developers can use their knowledge of the website or web application to help browsers single out those crucial resources more accurately.

This is what prefetching is, a hint for browsers to foresee users' every browsing wish and make it come true in a snap.

You can find prefetching in the [Resource Hints](#) specification led by Ilya Grigorik.

In what follows, I'm going to discuss:

- DNS-prefetching
- Link/Resource prefetching
- Prerendering (Page prefetching)

DNS-Prefetching

The internet is a network of computer-readable IP addresses (e.g., 87.87.87.87) that are referenced by human-readable hostnames (e.g., yoursite.com). DNS is the protocol that converts hostnames into IP addresses.

Each time the browser makes an HTTP request for a resource on a different domain, it can spend a few milliseconds to resolve the domain to the associated IP address.

If a website has a Twitter or Facebook widget, Google Analytics, and perhaps one or two custom web fonts, then it must have links to other domains. This makes DNS lookups unavoidable.

DNS prefetching helps to decrease waiting time for visitors because the browser performs DNS lookups before it sends a request for resources located on other domains.

So, let's say developers know a website will make a request to somewidget.example.com. They can drop a hint for the browser suggesting it to go ahead and prefetch that hostname's DNS by adding a `rel` attribute to the link with the value of `dns-prefetch`, like this:

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

Now, when the request to somewidget.example.com takes place, the browser has already performed the DNS lookup ahead of time and users get the results delivered a bit sooner.

Browser support for DNS-prefetch at this time is present in all major desktop browsers. When support for DNS-prefetching is lacking, browsers simply retrieve resources in the usual way. No big deal.

What if the browser never requests the DNS-prefetched resource? Thankfully, DNS-prefetching is not a costly operation, since it doesn't send more than just a few hundred bytes over the network. Once again, no harm done.

Link Prefetching

According to the [spec](#), link prefetching ...

is used to identify a resource that might be required by the next navigation, and that the user agent SHOULD fetch, such that the user agent can deliver a faster response once the resource is requested in the future.

In other words, if developers have grounds to assume users are likely to visit a specific web page and know the browser needs some critical resources to deliver it, they can use the `prefetch` directive. This points the browser towards the resources it should fetch before users actually navigate to that page.

Keep in mind that prefetching only works with cacheable resources, such as JavaScript, images and so on. Here's what the code looks like:

```
<link rel="prefetch" href="//example.com/future-image.jpg">
```

The browser now knows that the `future-image.jpg` image is going to be needed soon, so it can prefetch it and store it locally in the cache. At that point, because the browser will have already done the time-consuming work of downloading the resource in the background, the benefit of this technique consists in super-fast rendering as soon as users click to access the relevant page requiring `future-image.jpg`.

Link prefetching is great at creating the perception of a fast-loading website, but it's a much more expensive operation than DNS-prefetching. If users never

access the page which needs the prefetched asset, the browser will have downloaded unnecessary data and clogged up its cache.

At the time of writing, link prefetching is supported by the latest versions of Chrome, Firefox, IE/Edge and Opera. Non-supporting browsers will simply ignore the hint.

Page Prefetching/Prerendering

Implementing page prefetching or prerendering is just a matter of adding the prerender directive inside a link's `rel` attribute, like this:

```
<link rel="prerender" href="//example.com/future-page.html">
```

Prerendering goes all the way and literally creates an invisible version of the entire page users are likely to go to next, including applying CSS and executing JavaScript. As soon as users click on the relevant link, the ghost page instantly materializes before their eyes, replacing the old content.

If link prefetching can be an expensive operation, prerendering is even more so. In this case, browsers download an entire web page and all its resources on the basis of an expectation that users will visit that page, which ultimately may or may not happen.

At this time, prerendering is supported in IE/Edge, Chrome and Opera.

Use Cases for Link Prefetching and Prerendering

Since both link prefetching and (to a greater extent) prerendering don't come cheap, having a definite idea of when it's suitable to implement them on a web page is important.

Devs who plan on using resource hints should do so when they have solid grounds for assuming which resources to prefetch or prerender based on the website's content, users' behavior, analytics, etc.

The Resource Hints spec indicates a number of use cases where link prefetching could benefit user experience:

- Search results — It's reasonable to assume users are interested in those results, therefore it's likely they're going to click the links leading to the content they're after.
- Paginated content — Once users are reading the first page of a multi-page article, it's not too far-fetched to assume they're going to click the link that lands them on the next page.
- Image galleries — Google devs use prefetching on Picasa Web Albums. If users are viewing a photo, developers are justified in guessing that the next photo is also going to be accessed, so they instruct the browser to download it as soon as possible.

Similarly, Santiago Valdarrama offers some helpful tips on when developers could reasonably consider implementing prerendering on a website:

When deciding whether to prerender entire pages ahead of time, consider that Google prerenders the top results on its search page, and Chrome prerenders pages based on the historical navigation patterns of users. Using the same principle, you can detect common usage patterns and prerender target pages accordingly. You can also use it, just like resource prefetching, on questionnaires or surveys where you know users will complete the workflow in a particular order.

Resources

If you'd like to dig deeper, the following resources are a must-read:

- [Resource Hints Specification](#)

- [Web Performance Tricks – Beyond the Basics](#)
- [Prefetching resources](#)
- [Prefetching, preloading, prebrowsing](#)
- [One Step Ahead: Improving Performance with Prebrowsing](#)
- [Front-end performance for web designers and front-end developers](#)

Conclusion

DNS-prefetching, link prefetching and prerendering are powerful optimization techniques. If **implemented responsibly**, on the basis of the knowledge developers have of web content and user behavior, prefetching can considerably improve user experience.

Optimizing Web Fonts for Performance: the State of the Art

Maria Antonietta Perna

Chapter

6

This article is part of a series created in partnership with [SiteGround](#). Thank you for supporting the partners who make SitePoint possible.

67% of webpages now use custom fonts. However, popularity and widespread adoption haven't come without some performance and user experience-related issues.

In this article, I'll go through what's not so good about the way web fonts are commonly used and loaded, as well as point you to well-tested workarounds and future standards-based solutions.

Why Custom Web Fonts?

Users come to your website for content. Therefore, great typography is crucial on the web: readability, legibility and well-crafted typographic design are a must for brand recognition and the success of your message.

The best way to achieve beautiful typography is by loading custom web fonts --- that is, font files that are not already installed on users' devices.

Since browser support for the CSS `@font-face` rule has become widespread, using custom web fonts in websites has increased by leaps and bounds. However, fonts can have a heavy file size, and loading extra resources on your website doesn't come without some negative impact on performance.

Since file size can certainly be an issue, paying attention to how custom web fonts are loaded comes to the forefront.

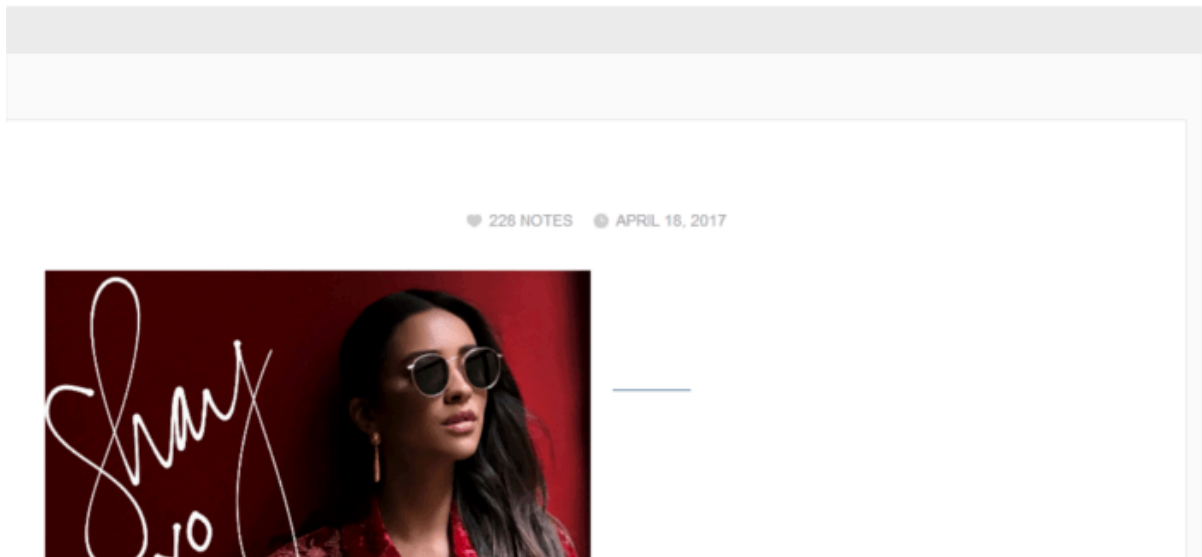
What Is the Flash of Invisible Text (FOIT) All About?

The most common way of using a custom web font is to specify the fonts used inside a CSS `@font-face` declaration and leave the browser to its default behavior. This is not ideal. Since information about fonts is located in the CSS, by default browsers delay the loading of fonts until the CSS is parsed. That's not all.

As Zach Leatherman has made abundantly clear, to trigger a font download, a number of conditions must be met besides a valid `@font-face` declaration:

- an HTML node that uses the same `font-family` specified inside `@font-face`
- in Webkit and Blink browsers, that node must not be empty
- in browsers that support the unicode range descriptor inside `@font-face`, the content must also match the specified unicode range.

If all the points above are satisfied, the browser starts downloading the font. This means that the browser needs to have parsed not only the CSS code but also a website's text content before it can trigger a font's download. However, it's just when the font starts loading that web users are likely to experience the dreaded Flash Of Invisible Text, or FOIT for short.



FOIT: Page at <http://blog.instagram.com/> while fonts are loading on Firefox v.52. Text is invisible.



FOIT: Page at <http://blog.instagram.com/> after fonts have been loaded on Firefox v.52. Text has become visible.

In other words, as soon as browsers start downloading a font, all text becomes invisible. So users look at a screen with no text for some time, which under sub-par network conditions can feel like forever. Furthermore, the way different browsers handle FOIT also varies:

- Blink and Firefox browsers tackle FOIT by introducing a three-second timeout: the text disappears for up to three seconds during font loading. If the font hasn't loaded within this time frame, the website displays a fallback font, which is subsequently replaced with the custom font once this is fully loaded. This behavior gives rise to what is known as **FOUT** (another acronym): **Flash Of Unstyled Text**.
- Safari, the default Android Browser and Blackberry don't use a timeout but display no text until the custom font is loaded. If anything goes wrong and the font doesn't get loaded, users are left with invisible text on the screen.
- IE/Edge browsers don't hide text, but display the fallback font right away, which seems to be a better design decision by Microsoft.

Although having a sudden change in typeface when reading text on a website is not the best user experience, staring at a blank screen while the font is loading sounds much worse. Ideally, a working approach should overcome both FOUT and FOIT. However, although it's an open issue, there is some consensus among a number of experts that FOUT is way better than FOIT.

Tackling the issues related to loading fonts involves working both on optimizing the file size and taking control of the default browser behavior so as to eliminate FOIT and minimizing the jarring impact of FOUT.

Let's look closer into each task one at a time.

Tips on Optimizing Custom Font Files

There are a few steps you can take to make sure your font files are not super heavy.

#1 Minimize the number of typefaces in your project

Don't think that just because you can choose from tons of beautiful typefaces, you need to weigh down your website with a multitude of different fonts. The opposite is the case.

In most situations, a couple of judiciously paired typefaces can work wonders for your design and have a lighter impact on website performance.

#2 Provide web font formats based on browser support

There are four main font formats:

- **True Type Font (TTF)**, a common font format that's been around since the late 80s
- **Web Open Font Format (WOFF)**, a format developed in 2009 which is Open Type or True Type, only with compression and further metadata

- **Web Open Font Format (WOFF2)**, a better compressed format than WOFF
- **Embedded Open Type (EOT)**, a format designed by Microsoft to be used for embedded fonts on the web.

Although you could specify all of them in your `@font-face` declaration, you can get away with just two of them. Here's how:

```
[code language="css"] @font-face { font-family: 'Open Sans'; src: local('Open Sans'), local('OpenSans'), url('fonts/open-sans.woff2') format('woff2'), url('fonts/open-sans.woff') format('woff'); } [/code]
```

The first format you suggest to the browser is `woff2`, which gives you the advantage of extra compression. If your browser doesn't support `woff2`, it just selects `woff`, which has good compression and enjoys support by the latest versions of all major browsers. Only Opera Mini lacks support for it, together with IE8 and older Android Mobile browsers. If you use a progressive enhancement approach to web development, you could simply let these older browsers fall back on a system font, e.g., Arial, Verdana, etc.

#3 Load just the styles you need

Fonts usually have different variants --- italic, bold, etc. Each font variation adds weight to the file size, so try to avoid adding a font variation to your `@font-face` code if you're not going to use it on your website. Keep in mind that if you use an `<i>` tag to add an icon, which is quite common, that is enough to get the browser to load the italic font variant (Slide 77 in Zack Leatherman's Santa Clara Velocity talk of 2015). If that's the only instance of `<i>` in your page, using a `` tag for the icon or setting the `font-style` for the `<i>` element to `normal` instead would spare the browser from having to download unnecessary resources.

#4 Keep your character sets down

Avoid loading all the characters, numbers and symbols your chosen font makes available. Instead, select just the character sets you actually use on your

webpage.

You can learn everything about fonts subsetting in [Slash Page Load Times With CSS Font Subsetting](#), by Dudley Storey.

Tackling FOIT

Let's look at some alternatives available today for dealing with FOIT. I've singled out the simplest approaches to implement among effective and recommended practices. However, you can learn about far more sophisticated ones in Zach Leatherman's [A Comprehensive Guide to Font Loading Strategies](#).

Don't use custom fonts

If you can't stand the idea of your users staring at invisible text or at font-swapping shifts, the most simple option is to give up on custom fonts and rely solely on web-safe fonts.

These fonts are preinstalled on users' devices, so the browser doesn't need to load them. Even when deciding to use custom fonts, web safe-fonts are commonly added to the [font stack](#) as a fallback.

Giving up on the perfect custom font for your design is not an exciting approach, but it's perfectly viable --- especially if you consider this as a transitory stance until cutting edge solutions, which are already on the horizon, gain wide browser support.

Opting for this approach doesn't require you to use `@font-face`. You just add your font stack to the CSS `font-family` property. Here's how you could do this today:

```
[code language="css"] body { font-family: -apple-system, BlinkMacSystemFont, "Segoe UI", Roboto, Oxygen-Sans, Ubuntu, Cantarell, "Helvetica Neue", sans-serif; } [/code]
```


[The New System Font Stack?](#), by Aderinokun, provides more information on the modern font stack.

The Web Font Loader

One long-standing solution is the [Web Font Loader](#), a JavaScript library developed by Google and [Typekit](#), which works with fonts coming from multiple sources --- such as Google Fonts, Typekit, your self-hosted font files, etc.

The Web Font Loader loads custom fonts as a background process while displaying fallback fonts to users, thereby avoiding FOIT. You can set a time limit of your choice within which the font must load. If the request exceeds this limit, users will only see the text styled with a fallback font. Once the custom font is loaded, the script swaps the fallback font on the page with the custom font.

Here are two awesome tutorials to learn how to use the Web Font Loader:

- [How to Improve Page Performance with a Font Loader](#)
- [Loading Web Fonts with the Web Font Loader](#)

The CSS Font Loading API

The [CSS Font Loading API](#) is a standards-based way of loading and taking control of web fonts.

The overall recommended approach remains that of avoiding FOIT and managing FOUT as best as possible. This API can keep track of every stage of font loading. You can use this information to style text using web-safe fonts while fonts are not available and style text using custom fonts once these are fully loaded.

At the time of writing, the latest versions of [Chrome, Firefox, Safari and Opera](#) support this API, while IE/Edge doesn't offer support yet. However, you can use a small JavaScript library, the [Font Face Observer](#), to polyfill the native CSS Font

Loading API, or simply let non-supporting browsers degrade gracefully to web-safe fonts.

For a short tutorial on how to use the CSS Font Loading API, head over to [Getting started with CSS Font Loading](#) by Manuel Matuzovic and have a look at his [CodePen demo](#).

The Future: The CSS `font-display` Property

Since their first appearance, you've handled custom fonts using CSS `@font-face` and `font-family`. It's only fair to expect CSS to solve any issue related to downloading custom fonts. But at present, only JavaScript-based solutions are usable.

However, this won't be so indefinitely. A smart CSS property is around the corner, `font-display`, which is part of the [CSS Font Rendering Controls Module Level 1](#).

`font-display` lets you:

- Decide if you want to show text using a fallback font or hide it while the font is loading.
- Control what you want to do once the font has loaded, i.e., continue to show the fallback font or replace it with the custom font. You can also do so on a per-element basis.
- Specify your own timeout values for each font and even for a specific element.

Here's what this property looks like in code:

```
[code language="css"] @font-face { font-family: Lato; src: url('/web/css/fonts/lato/lato-regular-webfont.woff2') format('woff2'), url('/web/css/fonts/lato/lato-regular-webfont.woff') format('woff'); font-weight: 400; font-style: normal; /* This value replaces fallback when font has loaded */ font-display: swap; } body { font-
```

```
family: Lato, sans-serif; font-weight: 400; font-style: normal; } [/code]
```

Accepted values for this property are:

- **auto**: browsers just implement their default behavior.
- **block**: browsers draw invisible text while the font is loading and replace it with the desired font as soon as it's loaded.
- **swap**: browsers use a fallback font while the custom font isn't available, but swap the custom font in as soon as it loads.
- **fallback**: similar behavior to **swap**. However, if too much time passes until the custom font loads, browsers use the fallback font for the duration of the page's lifetime.
- **optional**: this super smart value lets browsers use the custom font only if it's already available. If it isn't, browsers use the fallback for the duration of the page's lifetime. The custom font may still be downloaded in the background for use on subsequent page loads. However, if browsers detect limited bandwidth on users' devices, they might not load the custom font at all.

From these short descriptions, you can already see how **swap** and **optional** could be super useful.

Browser support for `font-display` is non-existent at this time, but it can be enabled in Chrome (via the "Experimental Web Platform features" flag) and Firefox (via the `layout.css.font-display.enabled` flag).

What About FOUT?

Although all the approaches outlined above make possible the elimination of FOIT, either by swapping in the custom font at a later time or by displaying just the fallback font, they can do little against FOUT.

Although FOUT makes text still available to users, it can very well be annoying, especially when the fallback font is wider and/or higher than the custom font it is temporarily replacing, which often causes a shift in the page content.

You can mitigate the jarring effect of the swap by adjusting the fallback font's x-height and width so as to match the desired custom font's x-height and width as closely as possible.

You can do this in a number of ways, but I find this little [Font Style Matcher tool](#) by Monica Dinculescu really helpful.

Conclusion

In this article I've outlined the issues of file size and Flash of Invisible Text related to using custom fonts on websites.

I've presented the latest approaches and pointed you to further resources where you can read more and learn how to implement those techniques in your projects.

JavaScript Performance Optimization Tips: An Overview

Ivan Čurić

Chapter

7

In this chapter, there's lots of stuff to cover across a wide and wildly changing landscape. It's also a topic that covers everyone's favorite: The JS Framework of the Month™.

We'll try to stick to the "Tools, not rules" mantra and keep the JS buzzwords to a minimum. Since we won't be able to cover everything related to JS performance in one or two chapters, make sure you read the references and do your own research afterwards.

But before we dive into specifics, let's get a broader understanding of the issue by answering the following: what is considered as performant JavaScript, and how does it fit into the broader scope of web performance metrics?

Setting the Stage

First of all, let's get the following out of the way: if you're testing exclusively on your desktop device, you're excluding more than 50% of your users.

The number of mobile users overtook the number of desktop users in November 2016

This trend will only continue to grow, as the emerging market's preferred gateway to the web is a sub-\$100 Android device. The era of the desktop as the main device to access the Internet is over, and the next billion internet users will visit your sites primarily through a mobile device.

Testing in Chrome DevTools' device mode isn't a valid substitute to testing on a real device. Using CPU and network throttling helps, but it's a fundamentally different beast. Test on real devices.

Even if you *are* testing on real mobile devices, you're probably doing so on your brand spanking new \$600 flagship phone. The thing is, that's not the device your users have. The median device is something along the lines of a Moto G1 --- a device with under 1GB of RAM, and a very weak CPU and GPU.

Let's see how it stacks up when parsing an average JS bundle.

JS parsing speed comparison across devices

Source: Addy Osmani - Time spent in JS parse & eval for average JS.

Ouch. While this image only covers the parse and compile time of the JS (more on that later) and not general performance, it's strongly correlated and can be treated as an indicator of general JS performance.

To quote Bruce Lawson, "it's the World-Wide Web, not the Wealthy Western Web". So, your target for web performance is a device that's ~25x slower than your MacBook or iPhone. Let that sink in for a bit. But it gets worse. Let's see what we're actually aiming for.

What Exactly Is Performant JS Code?

Now that we know what our target platform is, we can answer the next question: what *is* performant JS code?

While there's no absolute classification of what defines performant code, we do have a user-centric performance model we can use as a reference: The RAIL model.

RAIL: Respond/Animate/Idle work/Load

Source: Sam Saccone - Planning for Performance: PRPL

Respond

If your app responds to a user action in under 100ms, the user perceives the response as immediate. This applies to tappable elements, but not when scrolling or dragging.

Animate

On a 60Hz monitor, we want to target a constant 60 frames per second when animating and scrolling. That results in around 16ms per frame. Out of that 16ms budget, you realistically have 8–10ms to do all the work, the rest taken up by the browser internals and other variances.

Idle work

If you have an expensive, continuously running task, make sure to slice it into smaller chunks to allow the main thread to react to user inputs. You shouldn't have a task that delays user input for more than 50ms.

Load

You should target a page load in under 1000ms. Anything over, and your users start getting twitchy. This is a pretty difficult goal to reach on mobile devices as it relates to the page being interactive, not just having it painted on screen and scrollable. In practice, it's even less:

JavaScript 1000ms budget

[Fast By Default: Modern Loading Best Practices \(Chrome Dev Summit 2017\)](#)

In practice, aim for the 5s time-to-interactive mark. It's what Chrome uses in their [Lighthouse audit](#).

Now that we know the metrics, [let's have a look at some of the statistics](#):

- 53% of visits are abandoned if a mobile site takes more than three seconds to load
- 1 out of 2 people expect a page to load in less than 2 seconds
- 77% of mobile sites take longer than 10 seconds to load on 3G networks
- 19 seconds is the average load time for mobile sites on 3G networks.

And a bit more, courtesy of Addy Osmani:

- apps became interactive in 8 seconds on desktop (using cable) and 16 seconds on mobile (Moto G4 over 3G)
- at the median, developers shipped 410KB of gzipped JS for their pages.

Feeling sufficiently frustrated? Good. Let's get to work and fix the web. ?

Context is Everything

You might have noticed that the main bottleneck is the time it takes to load up your website. Specifically, the JavaScript download, parse, compile and execution time. There's no way around it but to load less JavaScript and load smarter.

But what about the actual work that your code does aside from just booting up the website? There has to be some performance gains there, right?

Before you dive into optimizing your code, consider what you're building. Are you building a framework or a VDOM library? Does your code need to do thousands of operations per second? Are you doing a time-critical library for handling user input and/or animations? If not, you may want to shift your time and energy somewhere more impactful.

It's not that writing performant code doesn't matter, but it usually makes little to no impact in the grand scheme of things, especially when talking about microoptimizations. So, before you get into a Stack Overflow argument about `.map` vs `.forEach` vs `for` loops by comparing results from JSperf.com, make sure to see the forest and not just the trees. 50k ops/s might sound 50× better than 1k ops/s on paper, but it won't make a difference in most cases.

Parsing, Compiling and Executing

Fundamentally, the problem of most non-performant JS is not running the code

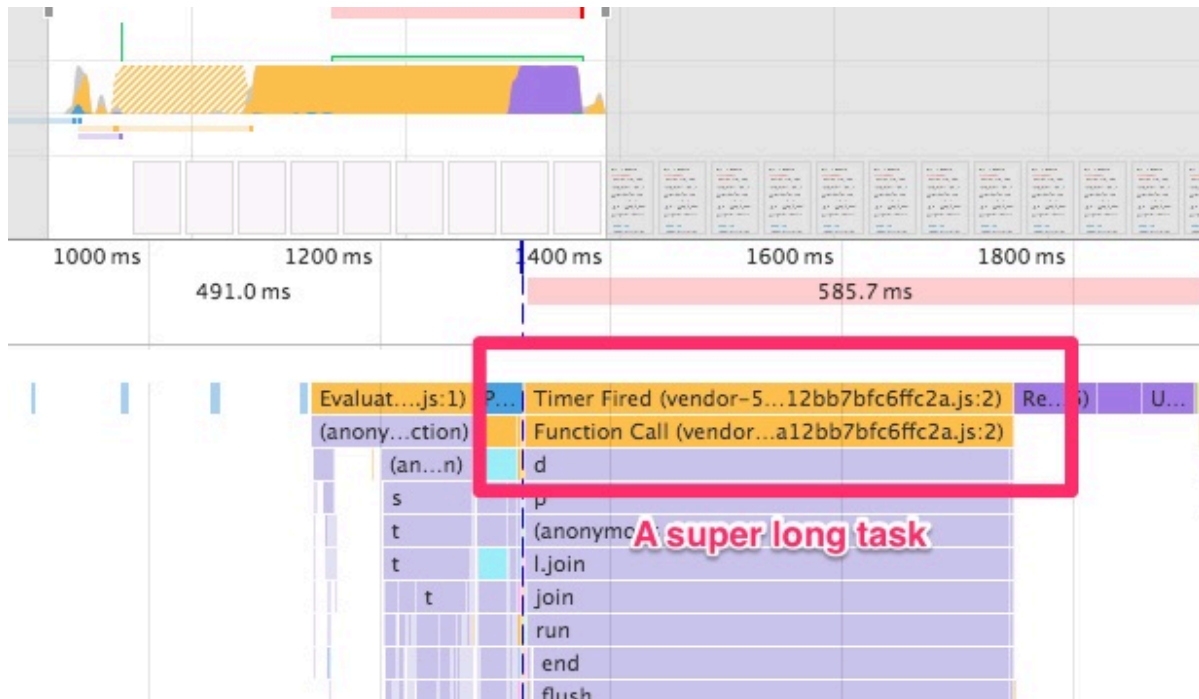
itself, but all the steps that have to be taken *before* the code even starts executing.

We're talking about levels of abstraction here. The CPU in your computer runs machine code. Most of the code you're running on your computer is in the compiled binary format. (I said *code* rather than *programs*, considering all the Electron apps these days.) Meaning, all the OS-level abstractions aside, it runs natively on your hardware, no prep-work needed.

JavaScript is not pre-compiled. It arrives (via a relatively slow network) as readable code in your browser which is, for all intents and purposes, the "OS" for your JS program.

That code first needs to be parsed --- that is, read and turned into an computer-indexable structure that can be used for compiling. It then gets compiled into bytecode and finally machine code, before it can be executed by your device/browser.

Another *very* important thing to mention is that JavaScript is single-threaded, and runs on the browser's main thread. This means that only one process can run at a time. If your DevTools performance timeline is filled with yellow peaks, running your CPU at 100%, you'll have long/dropped frames, janky scrolling and all other kind of nasty stuff.



Source: Paul Lewis - [When everything's important, nothing is!](#).

So there's all this work that needs to be done before your JS starts working. Parsing and compiling takes up to 50% of the total time of JS execution in Chrome's V8 engine.

Comparison of mobile and desktop JS parse time

Source: Addy Osmani - [JavaScript Start-up Performance](#).

There are two things you should take away from this section:

- 1 While not necessarily linearly, JS parse time scales with the bundle size.

The less JS you ship, the better.

- 2 Every JS framework you use (React, Vue, Angular, Preact...) is another level of abstraction (unless it's a precompiled one, like [Svelte](#)). Not only will it increase

your bundle size, but also slow down your code since you're not talking directly to the browser.

There are ways to mitigate this, such as using service workers to do jobs in the background and on another thread, using asm.js to write code that's more easily compiled to machine instructions, but that's a whole 'nother topic.

What you can do, however, is avoid using JS animation frameworks for everything and read up on what triggers paints and layouts. Use the libraries only when there's absolutely no way to implement the animation using regular CSS transitions and animations.

Even though they may be using CSS transitions, composited properties and `requestAnimationFrame()`, they're still running in JS, on the main thread. They're basically just hammering your DOM with inline styles every 16ms, since there's not much else they can do. You need to make sure all your JS will be done executing in under 8ms per frame in order to keep the animations smooth.

CSS animations and transitions, on the other hand, are running off the main thread --- on the GPU, if implemented performantly, without causing relayouts/reflows.

Considering that most animations are running either during loading or user interaction, this can give your web apps the much-needed room to breathe.

The Web Animations API is an upcoming feature set that will allow you to do performant JS animations off the main thread, but for now, stick to CSS transitions and techniques like FLIP.

Bundle Sizes are Everything

Today it's all about bundles. Gone are the times of Bower and dozens of `<script>` tags before the closing `</body>` tag.

Now it's all about `npm install`-ing whatever shiny new toy you find on NPM, bundling them together with Webpack in a huge single 1MB JS file and hammering your users' browser to a crawl while capping off their data plans.

Try shipping less JS. You might not need the entire Lodash library for your project. Do you absolutely *need* to use a JS framework? If yes, have you considered using something other than React, such as Preact or HyperHTML, which are less than 1/20 the size of React? Do you need TweenMax for that scroll-to-top animation? The convenience of npm and isolated components in frameworks comes with a downside: the first response of developers to a problem has become to throw more JS at it. When all you have is a hammer, everything looks like a nail.

When you're done pruning the weeds and shipping less JS, try shipping it *smarter*. Ship what you need, when you need it.

Webpack 3 has *amazing* features called code splitting and dynamic imports. Instead of bundling all your JS modules into a monolithic `app.js` bundle, it can automatically split the code using the `import()` syntax and load it asynchronously.

You don't need to use frameworks, components and client-side routing to gain the benefit of it, either. Let's say you have a complex piece of code that powers your `.mega-widget`, which can be on any number of pages. You can simply write the following in your main JS file:

```
if (document.querySelector('.mega-widget')) {  
    import('./mega-widget');  
}
```

If your app finds the widget on the page, it will dynamically load the required supporting code. Otherwise, all's good.

Also, Webpack needs its own runtime to work, and it injects it into all the .js files it generates. If you use the `commonChunks` plugin, you can use the following to extract the runtime into its own chunk:

```
new webpack.optimize.CommonsChunkPlugin({  
  name: 'runtime',  
}),
```

It will strip out the runtime from all your other chunks into its own file, in this case named `runtime.js`. Just make sure to load it before your main JS bundle. For example:

```
<script src="runtime.js">  
<script src="main-bundle.js">
```

Then there's the topic of transpiled code and polyfills. If you're writing modern (ES6+) JavaScript, you're probably using Babel to transpile it into ES5 compatible code. Transpiling not only increases file size due to all the verbosity, but also complexity, and it often has performance regressions compared to native ES6+ code.

Along with that, you're probably using the `babel-polyfill` package and `whatwg-fetch` to patch up missing features in older browsers. Then, if you're writing code using `async/await`, you also transpile it using generators needed to include the `regenerator-runtime`...

The point is, you add almost 100 kilobytes to your JS bundle, which has not only a huge file size, but also a huge parsing and executing cost, in order to support older browsers.

There's no point in punishing people who are using modern browsers, though. An approach I use, and which Philip Walton covered in [this article](#), is to create two separate bundles and load them conditionally. Babel makes this easy with `babel-preset-env`. For instance, you have one bundle for supporting IE 11, and the other without polyfills for the latest versions of modern browsers.

A dirty but efficient way is to place the following in an inline script:

```
(function() {  
  try {  
    new Function('async () => {}')();  
  } catch (error) {  
    // create script tag pointing to legacy-bundle.js;  
    return;  
  }  
  // create script tag pointing to modern-bundle.js;  
})();
```

If the browser isn't able to evaluate an `async` function, we assume that it's an old browser and just ship the polyfilled bundle. Otherwise, the user gets the neat and modern variant.

Conclusion

What we would like you to gain from this article is that JS is expensive and should be used sparingly.

Make sure you test your website's performance on low-end devices, under real network conditions. Your site should load fast and be interactive as soon as possible. This means shipping less JS, and shipping faster by any means necessary. Your code should always be minified, split into smaller, manageable bundles and loaded asynchronously whenever possible. On the server side, make

sure it has HTTP/2 enabled for faster parallel transfers and gzip/Brotli compression to drastically reduce the transfer sizes of your JS.

And with that said, I'd like to finish off with the following tweet:

So it takes a *lot* for me to get to this point. But seriously folks, time to throw out your frameworks and see how fast browser can be.

— Alex Russell (@slightlylate) September 15, 2016

7 Performance Tips for Jank- free JavaScript Animations

Maria Antonietta Perna

Chapter

8

The role of web animation has evolved from mere decorative fluff to serving concrete purposes in the context of user experience — such as providing visual feedback as users interact with your app, directing users' attention to fulfill your app's goals, offering visual cues that help users make sense of your app's interface, and so on.

To ensure web animation is up to such crucial tasks, it's important that motion takes place at the right time in a fluid and smooth fashion, so that users perceive it as aiding them, rather than as getting in the way of whatever action they're trying to pursue on your app.

One dreaded effect of ill-conceived animation is **jank**, which is explained on jankfree.org like this:

Modern browsers try to refresh the content on screen in sync with a device's refresh rate. For most devices today, the screen will refresh 60 times a second, or 60Hz. If there is some motion on screen (such as scrolling, transitions, or animations) a browser should create 60 frames per second to match the refresh rate. Jank is any stuttering, juddering or just plain halting that users see when a site or app isn't keeping up with the refresh rate.

If animations are janky, users will eventually interact less and less with your app, thereby negatively impacting on its success. Obviously, nobody wants that.

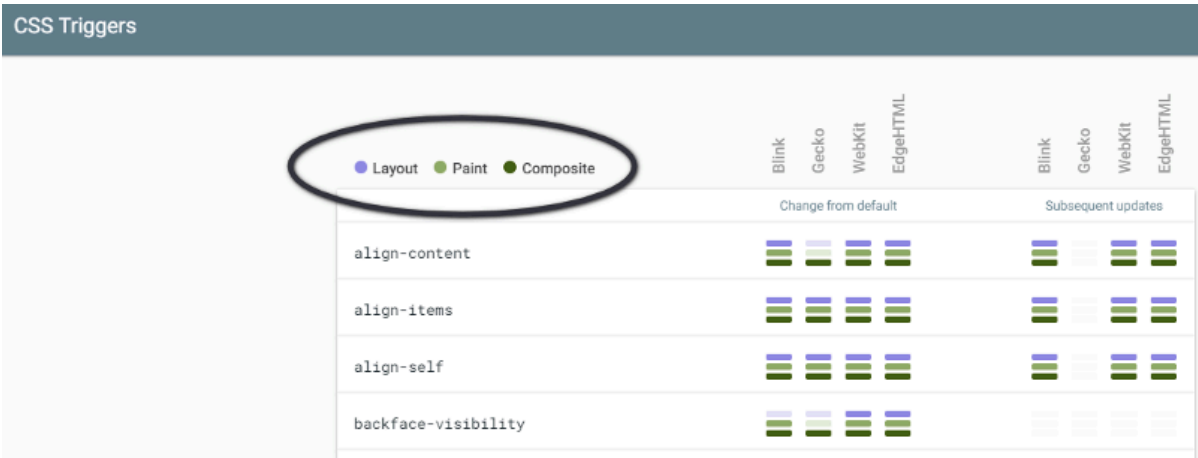
In this article, I've gathered a few performance tips to help you solve issues with JavaScript animations and make it easier to meet the 60fps (frame per second) target for achieving smooth motion on the web.

1. Avoid Animating Expensive CSS Properties

Whether you plan on animating CSS properties using CSS Transitions/CSS keyframes or JavaScript, it's important to know which properties bring about a change in the geometry of the page (layout) — meaning that the position of other

elements on the page will have to be recalculated, or that painting operations will be involved. Both layout and painting tasks are very expensive for browsers to process, especially if you have several elements on your page. As a consequence, you'll see animation performance improve significantly if you avoid animating CSS properties that trigger layout or paint operations and stick to properties like transforms and opacity, because modern browsers do an excellent job of optimizing them.

On [CSS Triggers](#) you'll find an up-to-date list of CSS properties with information about the work they trigger in each modern browser, both on the first change and on subsequent changes.



	Blink	Gecko	WebKit	EdgeHTML	Blink	Gecko	WebKit	EdgeHTML
	Change from default				Subsequent updates			
align-content	Layout	Layout	Layout	Layout	Layout	Layout	Layout	Layout
align-items	Layout	Layout	Layout	Layout	Layout	Layout	Layout	Layout
align-self	Layout	Layout	Layout	Layout	Layout	Layout	Layout	Layout
backface-visibility	Paint	Paint	Paint	Paint				

Changing CSS properties that only trigger composite operations is both an easy and effective step you can take to optimize your web animations for performance.

2. Promote Elements You Want to Animate to Their Own Layer (with Caution)

If the element you want to animate is on its own compositor layer, some modern browsers leverage hardware acceleration by offloading the work to the [GPU](#). If used judiciously, this move can have a positive effect on the performance of your

animations.

To have the element on its own layer, you need to **promote** it. One way you can do so is by using the CSS will-change property. This property allows developers to warn the browser about some changes they want to make on an element, so that the browser can make the required optimizations ahead of time.

However, it's not advised that you promote too many elements on their own layer or that you do so with exaggeration. In fact, every layer the browser creates requires memory and management, which can be expensive.

You can learn the details of how to use `will-change`, its benefits and downsides, in An Introduction to the CSS will-change Property by Nick Salloum.

3. Replace `setTimeout`/`setInterval` with `requestAnimationFrame`

JavaScript animations have commonly been coded using either `setInterval()` or `setTimeout()`.

The code would look something like this:

```
var timer;
function animateElement() {
  timer = setInterval( function() {
    // animation code goes here
  } , 2000 );
}

// To stop the animation, use clearInterval
function stopAnimation() {
```

```
clearInterval(timer);  
}
```

Although this works, the risk of jank is high, because the callback function runs at some point in the frame, perhaps at the very end, which can result in one or more frames being missed. Today, you can use a native JavaScript method which is tailored for smooth web animation (DOM animation, canvas, etc.), called `requestAnimationFrame()`.

`requestAnimationFrame()` executes your animation code at the most appropriate time for the browser, usually at the beginning of the frame.

Your code could look something like this:

```
function makeChange( time ) {  
  // Animation logic here  
  
  // Call requestAnimationFrame recursively inside the callback function  
  requestAnimationFrame( makeChange );  
}  
  
// Call requestAnimationFrame again outside the callback function  
requestAnimationFrame( makeChange );
```

[Performance with requestAnimationFrame](#) by Tim Evko here on SitePoint offers a great video introduction to coding with `requestAnimationFrame()`.

4. Decouple Events from Animations in Your Code

At 60 frames per second, the browser has 16.67ms to do its job on each frame.

That's not a lot of time, so keeping your code lean could make a difference to the smoothness of your animations.

Decoupling the code for handling events like scrolling, resizing, mouse events, etc., from the code that handles screen updates using `requestAnimationFrame()` is a great way to optimize your animation code for performance.

For a deep discussion of this optimization tip and related sample code, check out [Leaner, Meaner, Faster Animations with requestAnimationFrame](#) by Paul Lewis.

5. Avoid Long-running JavaScript Code

Browsers use the main thread to run JavaScript, together with other tasks like style calculations, layout and paint operations. Long-running JavaScript code could negatively impact on these tasks, which could lead to frames being skipped and janky animations as a consequence. Therefore, simplifying your code could certainly be a good way to ensure your animations run smoothly.

For complex JavaScript operations that don't require access to the DOM, consider using [Web Workers](#). The worker thread performs its tasks without impacting on the user interface.

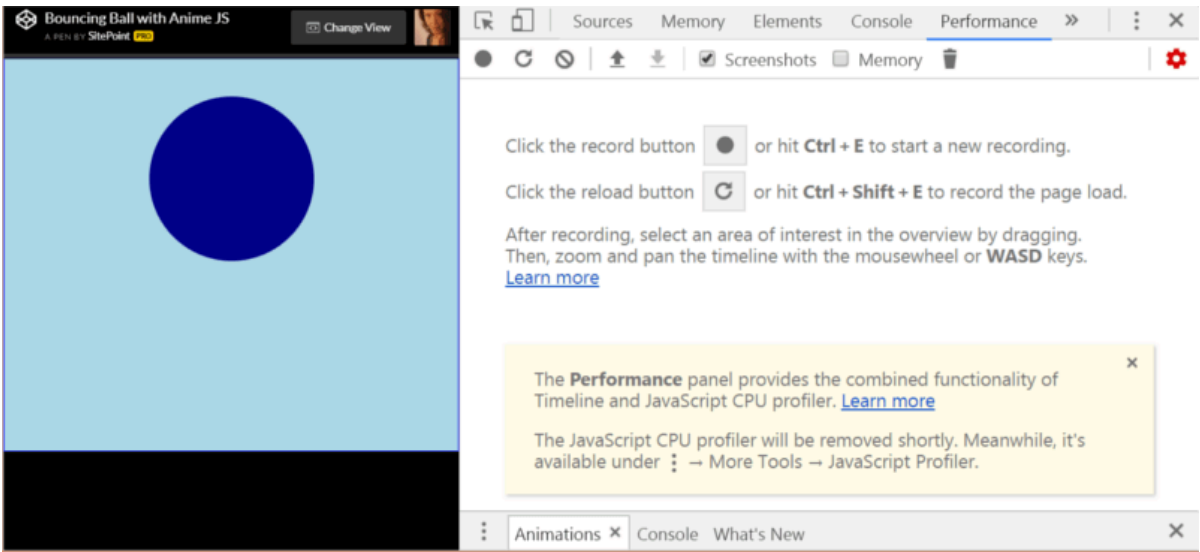
6. Leverage the Browser's DevTools to Keep

Performance Issues in Check

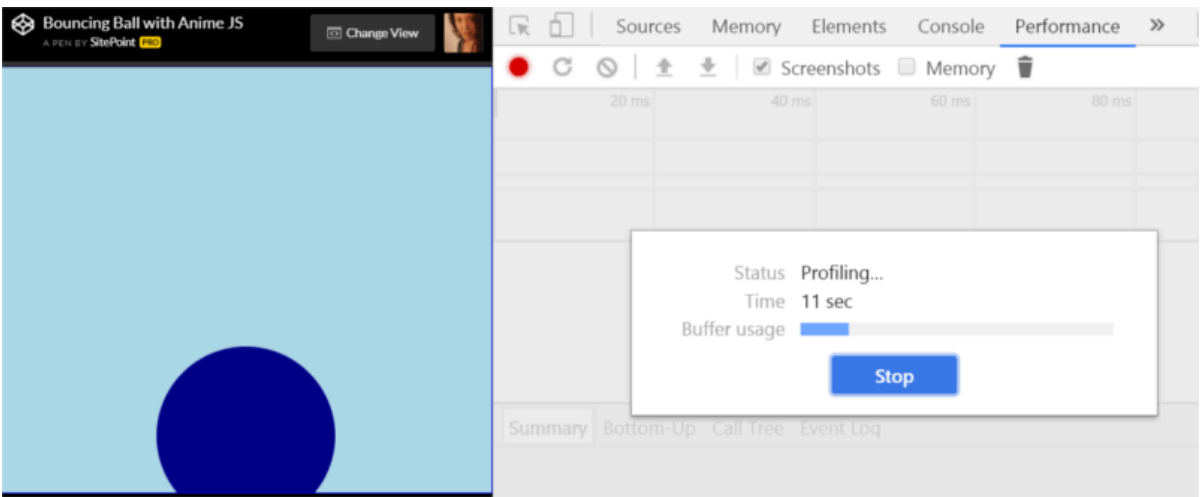
Your browser's developer tools provide a way to monitor how hard your browser is working to run your JavaScript code, or that of a third-party library. They also provide useful information about frame rates and much more.

You can access the Chrome DevTools by right-clicking on your web page and selecting *Inspect* inside the context menu. For example, recording your web page using the Performance tools will give you an insight into the performance

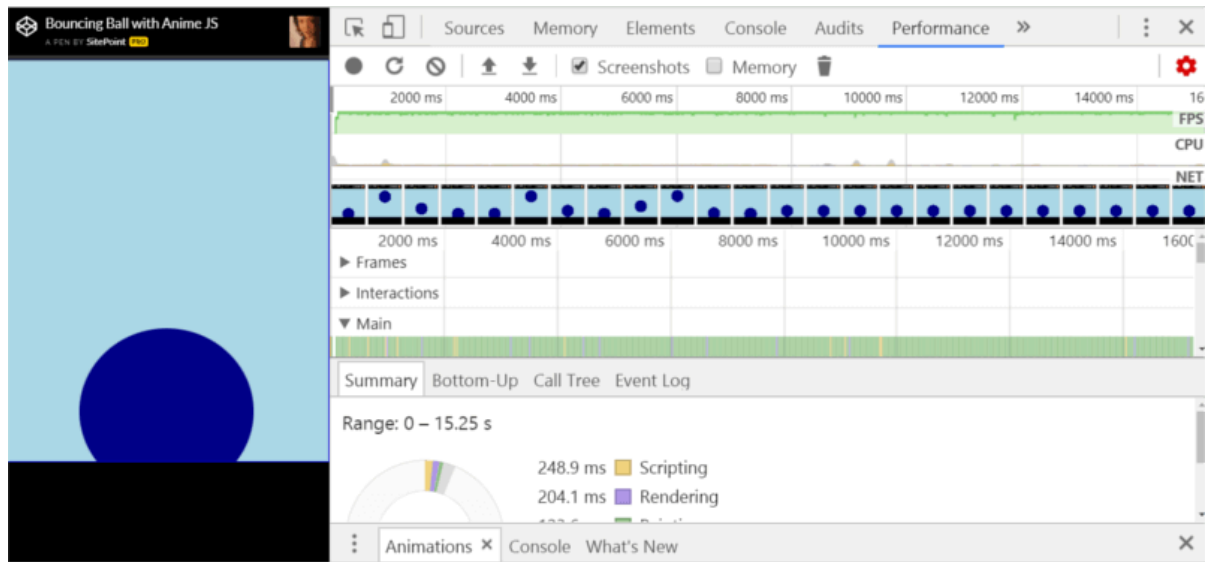
bottlenecks on that page:



Click the *record* button, then stop the recording after a few seconds:



At this point, you should have tons of data to help you analyze your page's performance:



This [Chrome DevTools Guide](#) will help you get the most out of DevTools for analyzing performance and lots of other kinds of data in your Chrome browser. If Chrome isn't your browser of choice, it's no big deal, as most modern browsers nowadays ship with super powerful DevTools that you can leverage to optimize your code.

7. Use an Off-screen Canvas for Complex Drawing Operations

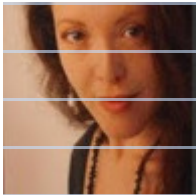
This tip relates specifically to optimizing code for [HTML5 Canvas](#).

If your frames involve complex drawing operations, a good idea would be to create an off-screen canvas where you perform all the drawing operations once or just when a change occurs, and then on each frame just draw the off-screen canvas.

You can find the details and code samples relative to this tip and lots more in the [Optimizing Canvas](#) article on MDN.

Conclusion

Optimizing code for performance is a necessary task if you don't want to fail users' expectations on the web today, but it's by no means always easy or straightforward. There might be several reasons why your animations aren't performing well, but if you try out the tips I listed above, you'll go a long way towards avoiding the most common animation performance pitfalls, thereby improving the user experience of your website or app.



Maria Antonietta Perna is co-Editor of the HTML/CSS Channel at SitePoint and a front-end web developer. She enjoys tinkering with cool CSS standards and is curious about teaching approaches to front-end code. When not coding for the web or not writing for the web, she enjoys philosophy books, long walks and good food.