



C

# Performance Optimization: Techniques And Strategies

# Imprint

© 2014 Smashing Magazine GmbH, Freiburg, Germany

ISBN (PDF): 978-3-94454087-0

Cover Design: Veerle Pieters

eBook Strategy and Editing: Vitaly Friedman

Technical Editing: Cosima Mielke

Planning and Quality Control: Vitaly Friedman, Iris Lješnjanin

Tools: Elja Friedman

Syntax Highlighting: Prism by Lea Verou

Idea & Concept: Smashing Magazine GmbH

## *About This Book*

Slow loading times break the user experience of any website — no matter how well crafted it might be. In fact, it only takes three seconds until users lose their interest in a site if they don't get a response immediately. If another site happens to be 250ms faster than yours, then users are more inclined to switch to a competitor's website in no time. Web fonts, heavy JavaScript, third-party widgets — all of them can sum up to become a real performance bottleneck. Nevertheless, tracking that down does not only improve loading times but also results in a much snappier experience and a higher user engagement.

In this eBook, we've compiled an entire selection of front-end and server-side techniques that will help you tackle such bottlenecks. Find out how to speed up existing websites, build high-performance sites (for both mobile and desktop), and prepare them for heavy-load situations. Furthermore, you'll learn more about how performance improvements and a 97–99 Google PageSpeed score were achieved on Smashing Magazine, as well as how optimization strategies can enhance real-life projects by taking a closer look at Pinterest's paint performance case study. With the help of this eBook, you'll notice that it's high time to dig deeper into your own site and examine it closely. Why don't you polish it up and make it even better than it already is!

## TABLE OF CONTENTS

Improving Smashing Magazine's Performance: A Case Study .....	5
How To Speed Up Your WordPress Website .....	50
You May Be Losing Users If Responsive Web Design Is Your Only Mobile Strategy .....	64
How To Make Your Websites Faster On Mobile Devices .....	86
Creating High-Performance Mobile Websites .....	115
Don't Get Crushed By The Load: Optimization Techniques And Strategies .....	137
Speed Up Your Mobile Website With Varnish .....	157
Cache Invalidation Strategies With Varnish Cache .....	169
Gone In 60 Frames Per Second: A Pinterest Paint Performance Case Study .....	179
About The Authors .....	205



# Improving Smashing Magazine's Performance: A Case Study

BY VITALY FRIEDMAN 🍷

Improvement is a matter of steady, ongoing iteration. When we redesigned Smashing Magazine back in 2012, our main goal was to establish trustworthy branding that would reflect the ambitious editorial direction of the magazine. We did that primarily by focusing on crafting a delightful reading experience. Over the years, our focus hasn't changed a bit; however, that very asset that helped to establish our branding turned into a major performance bottleneck.

## *Good Old-Fashioned Website Decay*

Looking back at the early days of our redesign, some of our decisions seem to be quick'n'dirty fixes rather than sound long-term solutions. Our advertising constraints pushed us to compromises. Legacy browsers drove us to dependencies on (relatively) heavy JavaScript libraries. Our technical infrastructure led us to heavily customized WordPress plugins and complex PHP logic. With every new feature added, our technical debt grew, and our style sheets, markup and JavaScript weren't getting any leaner.

Sound familiar? Admittedly, responsive web design as a technique often gets a pretty bad rap for bloating websites and making them difficult to maintain. (Not that

non-responsive websites are any different, but that's another story.) In practice, all assets on a responsive website will show up pretty much everywhere<sup>1</sup>: be it a slow smartphone, a quirky tablet or a fancy laptop with a Retina screen. And because media queries merely provide the ability to *respond* to screen dimensions – and do not, rather, have a more local, self-contained scope – adding a new feature and adjusting the reading experience potentially means going through each and every media query to prevent inconsistencies and fix layout issues.

## **“MOBILE FIRST” MEANS “ALWAYS MOBILE FIRST”**

When it comes to setting priorities for the content and functionality on a website, “mobile first” is one of those difficult yet incredibly powerful constraints that help you focus on what really matters, and identify critical components of your website. We discovered that designing mobile first is one thing; building mobile first is an entirely different story. In our case, both the design and development phases were heavily mobile first, which helped us to focus tightly on the content and its presentation. But while the design process was quite straightforward, implementation proved to be quite difficult.

Because the entire website was built mobile first, we quickly realized that adding or changing components on the page would entail going through the mobile-first ap-

---

1. <http://www.guypo.com/mobile/performance-implications-of-responsive-design-book-contribution/>

proach for every single (minor and major) design decision. We'd design a new component in a mobile view first, and then design an "extended" view for the situations when more space is available. Often that meant adjusting media queries with every single change, and more often it meant adding new stuff to style sheets and to the markup to address new issues that came up.



*Shortly after the new SmashingMag redesign went live, we ran into performance issues. An article by Tim Kadlec from 2012<sup>2</sup> shows just that.*

We found ourselves trapped: development and maintenance were taking a lot of time, the code base was full of minor and major fixes, and the infrastructure was becoming too slow. We ended up with a code base that had be-

---

<sup>2</sup>. <http://timkadlec.com/2012/01/work-to-be-done/>

come bloated before the redesign was even released — very bloated<sup>3</sup>, in fact.

## *Performance Issues*

In mid-2013, our home page weighed 1.4 MB and produced 90 HTTP requests. It just wasn't performing well. We wanted to create a remarkable reading experience on the website while avoiding the *flash of unstyled text* (FOUT), so web fonts were loaded in the header and, hence, were blocking the rendering of content (actually it's correct behaviour according to the spec<sup>4</sup>, designed to avoid multiple repaints and reflows.) jQuery was required for ads to be displayed, and a few JavaScripts depended on jQuery, so they all were blocking rendering as well. Ads were loaded and rendered *before* the content to ensure that they appeared as quickly as possible.

Images delivered by our ad partners were usually heavy and unoptimized, slowing down the page further. We also loaded Respond.js and Modernizr to deal with legacy browsers and to enhance the experience for smart browsers. As a result, articles were almost inaccessible on slow and unstable networks, and the start rendering time on mobile was disappointing at best.

It wasn't just the front-end that was showing its age though. The back-end wasn't getting any better either. In 2012 we were playing with the idea of having fully independent sections of the magazine — sections that would

---

3. <http://timkadlec.com/2012/01/work-to-be-done/>

4. <http://www.w3.org/TR/resource-priorities/#intro-download-priority>

live their own lives, evolving and growing over time as independent WordPress installations, with custom features and content types that wouldn't necessarily be shared across all sections.

Browser	Visitors		Filter results...	
<b>Google Chrome</b>	3,843,946	52.2%	<div></div>	
☆ <a href="#">Google Chrome 36.0</a>	2,484,089	33.7%	<div></div>	+999%
☆ <a href="#">Google Chrome 35.0</a>	750,265	10.2%	<div></div>	-63%
☆ <a href="#">Google Chrome 37.0</a>	341,849	4.6%	<div></div>	+999%
<b>Firefox</b>	1,326,491	18%	<div></div>	
☆ <a href="#">Firefox 31.0</a>	751,590	10.2%	<div></div>	+999%
<b>Mobile</b>	913,461	12.4%	<div></div>	
☆ <a href="#">Safari 7.0 mobile</a>	334,176	4.5%	<div></div>	+17%
☆ <a href="#">Google Chrome 36.0 mobile</a>	134,512	1.8%	<div></div>	+999%
<b>Safari</b>	620,166	8.4%	<div></div>	
☆ <a href="#">Safari 7.0</a>	404,202	5.5%	<div></div>	+25%
☆ <a href="#">Safari 5.1</a>	70,679	1%	<div></div>	+4%
<b>Internet Explorer</b>	348,388	4.7%	<div></div>	
☆ <a href="#">Internet Explorer 11.0</a>	161,108	2.2%	<div></div>	+28%
☆ <a href="#">Internet Explorer 8.0</a>	67,966	0.9%	<div></div>	-20%
☆ <a href="#">Internet Explorer 9.0</a>	67,160	0.9%	<div></div>	-11%
☆ <a href="#">Internet Explorer 10.0</a>	52,154	0.7%	<div></div>	+11%
<b>Opera</b>	73,870	1%	<div></div>	
☆ <a href="#">Opera 23.0</a>	34,040	0.5%	<div></div>	+999%

*Yes, we do enjoy a quite savvy user base, so optimization for IE8 is really not an issue.*

Because WordPress multi-install wasn't available at the time, we ended up with six independent, autonomous WordPress installs with six independent, autonomous style sheets. Those installs were connected to  $6 \times 2$  data-

bases (a media server and a static content server). We ran into dilemmas. For example, what if an author wrote for two sections and we'd love to show their articles from both sections on one single author's bio page? Well, we'd need to somehow pull articles from both installs and add redirects for each author's page to that one unified page, or should we just be using one of those pages as a "host"? Well, you know where this is going: increasing complexity and increasing maintenance costs. In the end, the sections didn't manage to evolve significantly — at least not in terms of content — yet we had already customized technical foundation of each section, adding to the CSS dust and PHP complexity.

(Because we had outsourced WordPress tasks, some plugins depended on each other. So, if we were to deactivate one, we might have unwittingly disabled two or three others in the process, and they would have to be turned back on in a particular order to work properly. There were even differences in the HTML outputted by the PHP templates behind the curtains, such as classes and IDs that differed from one installation to the next. It's no surprise that this setup made development a bit frustrating.)

The traffic was stagnant, readers kept complaining about the performance on the site and only a very small portion of users visited more than 2 pages per visit. The visual feedback when browsing the site was *visible* and surely wasn't instant, and this lag has been driving readers away from the site to Instapaper and Pocket — both on mobile and desktop. We knew that because we asked our

readers, and the feedback was quite clear (and a bit frustrating).

It was time to push back — heavily, with a major refactoring of the code base. We looked closely under the hood, discovering a few pretty scary (and nasty) things, and started fixing issues, one by one. It took us quite a bit of time to make things right, and we learned quite a few things along the way.

## *Switching Gears*

Up until mid-2013, we weren't using a CSS preprocessor, nor any build tools. Good long-term solutions require a good long-term foundation, so the first issues we tackled were tooling and the way the code base was organized. Because a number of people had been working on the code base over the years, some things proved to be rather mysterious... or challenging, to say the least.

We started with a code inventory, and we looked thoroughly at every single class, ID and CSS selector. Of course, we wanted to build a system of modular components, so the first task was to turn our seven large CSS files into maintainable, well-documented and easy-to-read modules. At the time, we'd chosen LESS, for no particular reason, and so our front-end engineer Marco<sup>5</sup> started to rewrite CSS and build a modular, scalable architecture. Of course, we could very well have used Sass instead, but Marco felt quite comfortable with LESS at the time.

---

<sup>5</sup>. <https://twitter.com/nice2meatu>

With a new CSS architecture, [Grunt](#)<sup>6</sup> as a build tool and a [few](#)<sup>7</sup> [time-saving](#)<sup>8</sup> [Grunt](#)<sup>9</sup> [tasks](#)<sup>10</sup>, the task of maintaining the entire code base became much easier. We set up a brand new testing environment, synced up everything with GitHub, assigned roles and permissions, and started digging. We rewrote selectors, reauthored markup, and refactored and optimized JavaScript. And yes, it took us quite some time to get things in order, but it really wouldn't have been so difficult if we hadn't had a number of very different stylesheets to deal with.

## THE BIG BACK-END CLEANUP

With the introduction of Multisite, creating a single WordPress installation from our six separate installations became a necessary task for our friends at [Inpsyde](#)<sup>11</sup>. Over the course of five months, Christian Brückner and Thomas Herzog cleaned up the PHP templates, kicked unnecessary plugins into orbit, rewrote plugins we had to keep and added new ones where needed. They cleared the databases of all the clutter that the old plugins had created — one of the databases weighed in at 70 GB (no, that's not a typo; we do mean gigabytes) — merged all of the databases into one, and then created a single fresh and, most importantly, *maintainable* WordPress Multisite installation.

---

6. <http://www.smashingmagazine.com/2013/10/29/get-up-running-grunt/>

7. <https://github.com/gruntjs/grunt-contrib-less>

8. <https://github.com/nDmitry/grunt-autoprefixer>

9. <https://github.com/gruntjs/grunt-contrib-cssmin>

10. <https://github.com/gruntjs/grunt-contrib-watch>

11. <http://inpsyde.com/en/>



The speed boost from those optimizations was remarkable. We are talking about 400 to 500 milliseconds of improvement by avoiding sub-domain redirects and unifying the code base and the back-end code. Those redirects<sup>12</sup> are indeed a major performance culprit, and just avoiding them is one of those techniques that usually boost performance significantly because you avoid full DNS lookups, improve time to first byte and reduce round trips on the network.

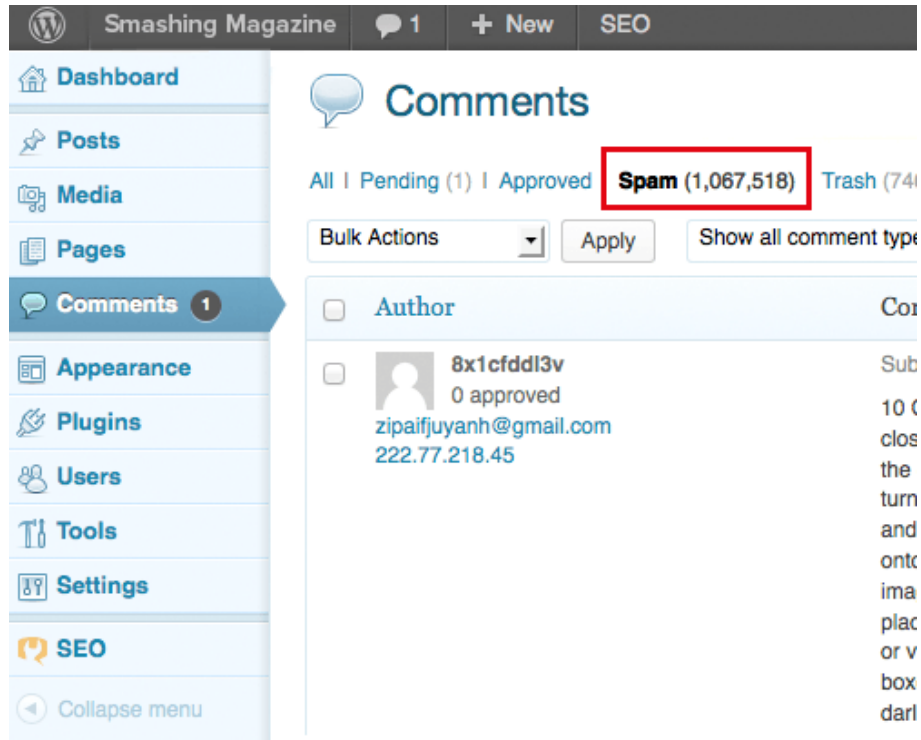
Thomas and Christian also refactored our entire WordPress theme according to the coding standard of their own theme architecture, which is basically a sophisticated way of writing PHP based on the WordPress standard. They wrote custom drop-ins that we use to display content at certain points in the layout. Writing the PHP strictly according to WordPress' official API felt like getting out of a horse-drawn carriage and into a race car. All modifications were done without ever touching WordPress' core, which is wonderful because we'll never have to fear updating WordPress itself anymore.

We migrated the installations during a slow weekend in mid-April 2014. It was a huge undertaking, and our server had a few hiccups during the process. We brought together over 2500 articles, including about 15,000 images, all spread over six databases, which also had a few major inconsistencies. While it was a very rough start at first – a lot of redirects had to be set up, caching issues on our server piled up, and some articles got lost between

---

<sup>12.</sup> <https://twitter.com/markodugonjic/statuses/478980625215782912>

the old and new installations — the result was well worth the effort.



*We've also marked a few millions spam comments across all the sections of the magazine. And before you ask: no, we did not import them into the new install.*

Our editorial team, primarily [Iris<sup>13</sup>](#), [Melanie<sup>14</sup>](#) and [Markus<sup>15</sup>](#), worked very hard to bring those lost articles back to life by analyzing our 404s with Google Webmaster Tools. We spent a few weekends to ensure that every single article was recovered and remains accessible. Losing articles, including their comments, was simply unacceptable.

<sup>13</sup>. [https://twitter.com/smash\\_it\\_on](https://twitter.com/smash_it_on)

<sup>14</sup>. [https://twitter.com/mel\\_in\\_media](https://twitter.com/mel_in_media)

<sup>15</sup>. <https://twitter.com/indysigner>

We know well how much time it takes for a good article to get published, and we have a lot of respect for authors and their work, and ensuring that the content remains online was a matter of respect for the work published. It took us a few weeks to get there and it wasn't the most enjoyable experience for sure, but we used the opportunity to introduce more consistency in our information architecture and to adjust tags and categories appropriately. (Ah, if you do happen to find an article that has gotten lost along the way, please do let us know<sup>16</sup> and we'll fix it right away. Thanks!)

## *Front-End Optimization*

In April 2014, once the new system was in place and had been running smoothly for a few days, we rewrote the LESS files based on what was left of all of the installs. Streamlining the classes for posts and pages, getting rid of all unneeded IDs, shortening selectors by lowering their specificity, and rooting out anything in the CSS we could live without crunched the CSS from 91 KB down to a mere 45 KB.

Once the CSS code base was in proper shape, it was time to reconsider how assets are loaded on the page and how we can improve the start rendering time beyond having clean, well-structured code base. Given the nightmare we experienced with the back-end previously, you might assume that improving performance now would have been a complex, time-consuming task, but actually it

---

<sup>16</sup>. <http://www.twitter.com/smashingmag>

was quite a bit easier than that. Basically, it was just a matter of getting our priorities right by optimizing the critical rendering path.

The key to improving performance was to focus on what matters most: the content, and the fastest way for readers to actually start reading our articles on their devices. So over a course of a few months we kept reprioritizing. With every update, we introduced mini-optimizations based on a very simple, almost obvious principle: optimize the delivery of content, and defer the rest — without any compromises, anywhere.

Our optimizations were heavily influenced by the work done by Scott Jehl<sup>17</sup>, as well as The Guardian<sup>18</sup> and the BBC<sup>19</sup> teams (both of which open-sourced their work). While Scott has been sharing valuable insight<sup>20</sup> into the front-end techniques that Filament Group was using, the BBC and The Guardian helped us to define and refine the concept of the core experience on the website and use it as a baseline. A shared main goal was to deliver the content as fast as possible to as many people as possible regardless of their device or network capabilities, and enhance the experience with progressive enhancement for capable browsers.

However, historically we haven't had a lot of JavaScript or complex interactions on Smashing Magazine, so we didn't feel that it was necessary to introduce

---

<sup>17</sup>. <https://github.com/scottjehl>

<sup>18</sup>. <https://github.com/guardian>

<sup>19</sup>. <https://github.com/BBC-News>

<sup>20</sup>. <http://filamentgroup.com/lab/performance-rwd.html>

complex loading logic with JavaScript preloaders. However, being a content-focused website, we *did* want to reduce the time necessary for the articles to start displaying as far as humanly possible.

## PERFORMANCE BUDGET: SPEED INDEX <= 1000

How fast is fast enough?<sup>21</sup> Well, that's a tough question to answer. In general, it's quite difficult to visualize performance and explain why every millisecond counts — unless you have hard data.



A nice way of visualizing performance is to use WebPageTest to generate an actual video of the page loading and run a test between two competing websites. Besides, the Speed Index metric<sup>22</sup> often proves to be very useful.

At the same time, falling into trap of absolutes and relying on not truly useful performance metrics is easy. In the past, the most commonly cited performance metric

---

<sup>21</sup>. <http://timkadlec.com/2014/01/fast-enough/>

<sup>22</sup>. <https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index>

was average loading time. However, on its own, average loading time isn't *that* helpful because it doesn't tell you much about when a user can actually start using the website. This is why talking about "fast enough" is often so tricky.

Different components require different amounts of time to load, yet some components of the page are more important than others. E.g. you don't need to load the *footer content* fast, but it's a good idea to render the visible portion of the page fast. You know where it's heading: of course, we are talking about the "above the fold" view here. As Ilya Grigorik once said<sup>23</sup>, "We don't need to render the entire page in one second, [just] the above the fold content." To achieve that, according to Scott's research and Google's test results, it's helpful to set ambitious performance goals:

- On WebPageTest<sup>24</sup>, aim for a Speed Index<sup>25</sup> value of under 1000.
- Ensure that all HTML, CSS and JavaScript fit within the first 14 KB.

What does it mean and why are they important? According to HCI research, "for an application to feel instant, a perceptible response to user input must be provided within hundreds of milliseconds<sup>26</sup>. After a second or more, the


---

<sup>23</sup>. <http://www.lukew.com/ff/entry.asp?1756>

<sup>24</sup>. <http://www.webpagetest.org/>

<sup>25</sup>. <https://sites.google.com/a/webpagetest.org/docs/using-webpagetest/metrics/speed-index>

user's flow and engagement with the initiated task feels broken.” With the first goal, we are trying to ensure an instant response on our website. It refers to the so-called *Speed Index* metric for the *start rendering* time — the average time (in ms) at which visible parts of the page are displayed, or become accessible. So the first goal basically reflects that a page starts rendering under 1000ms, and yes, it's a quite difficult challenge to take on.




## High Performance Browser Networking

by Ilya Grigorik

~~\$29.99~~ Read Online for Free  
Brought to you by Velocity Conference

Buy the Ebook on [oreilly.com](http://oreilly.com)

### About the Author



Ilya Grigorik is a web performance engineer and developer advocate on the Make The Web Fast team at Google, where he spends his days and nights on making the web fast and driving adoption of performance best practices. You can find Ilya online on his blog at [igvita.com](http://igvita.com) and on Twitter and Google+.

Ilya Grigorik's book *High Performance Browser Networking*<sup>27</sup> is a very helpful guide with useful guidelines and advice on making websites fast. And it's available as a free HTML book, too.

---

<sup>26</sup>. [http://chimera.labs.oreilly.com/books/1230000000545/ch10.html#SPEED\\_PERFORMANCE\\_HUMAN\\_PERCEPTION](http://chimera.labs.oreilly.com/books/1230000000545/ch10.html#SPEED_PERFORMANCE_HUMAN_PERCEPTION)

<sup>27</sup>. <http://chimera.labs.oreilly.com/books/1230000000545>

The second goal can help in achieving the first one. The value of 14 KB has been measured empirically<sup>28</sup> by Google and is the threshold for the first package exchanged between a server and client via towers on a cellular connection. You don't need to include images within 14 Kb, but you might want to deliver the markup, style sheets and any JavaScript required to render the visible portion of the page in that threshold. Of course, in practice this value can only realistically be achieved with gzip compression.

By combining the two goals, we basically defined a performance budget that we set for the website — a threshold for what was acceptable. Admittedly, we didn't concern ourselves with the start rendering time on different devices on various networks, mainly because we really wanted to push back as far as possible everything that isn't required to start rendering the page. So, the ideal result would be a Speed Index value that is *way* lower than the one we had set — as low as possible, actually — in all settings and on all connections, both shaky and stable, slow and fast. This might sound naive, but we wanted to figure out how fast we *could* be, rather than how fast we *should* be. We did measure start rendering time for first and subsequent page loads, but we did that much later, after optimizations had already been done, and just to keep track of issues on the front-end.

Our next step would be to integrate Tim Kadlec's Perf-Budget Grunt task<sup>29</sup> to incorporate the performance bud-

---

<sup>28</sup>. <https://www.youtube.com/watch?v=YV1nKLWoARQ>

<sup>29</sup>. <http://timkadlec.com/2014/05/performance-budgeting-with-grunt/>



get right into the build process and, thus, run every new commit against WebPagetest's performance benchmark. If it fails, we know that a new feature has slowed us down, so we probably have to reconsider how it's implemented to fit it within our budget, or at least we know where we stand and can have meaningful discussions about its impact on the overall performance.

## PRIORITIZATION AND SEPARATION OF CONCERNS

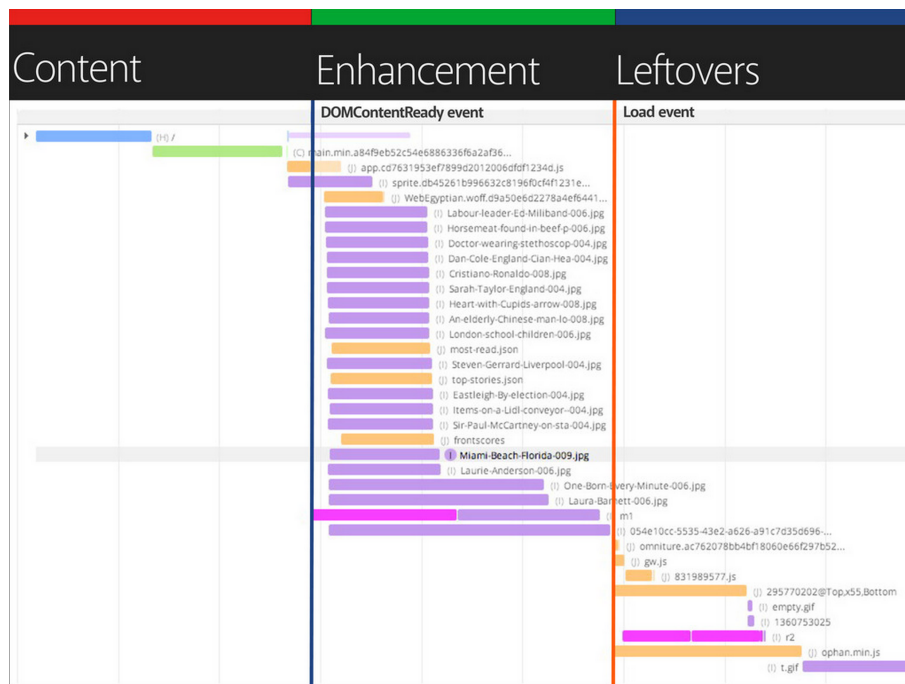
If you've been following *The Guardian's* work recently, you might be familiar with the strict separation of concerns that they introduced<sup>30</sup> during the major 2013 redesign. The Guardian separated<sup>31</sup> its entire content into three main groups:

- **Core content**  
Essential HTML and CSS, usable non-JavaScript-enhanced experience
- **Enhancement**  
JavaScript, geolocation, touch support, enhanced CSS, web fonts, images, widgets
- **Leftovers**  
Analytics, advertising, third-party content

---

<sup>30</sup>. <https://speakerdeck.com/andyhume/anatomy-of-a-responsive-page-load-whiskyweb-2013>

<sup>31</sup>. <https://vimeo.com/77967591>



*A strict separation of concerns, or loading priorities, as defined by The Guardian team.*

Once you have defined, confirmed and agreed upon these priorities, you can push performance optimization quite far. Just by being very specific about each type of content you have and by clearly defining what “core content” is, you are able to load *Core content* as quickly as possible, then load *Enhancements* once the page starts rendering (after the **DOMContentLoaded** event fires), and then load *Leftovers* once the page has fully rendered (after the **Load** event fires).

The main principle here of course is to strictly separate the loading of assets throughout these three phases, so that the loading of the *Core content* should never be blocked by any resources grouped in *Enhancement* or *Leftovers* (we haven’t achieved the perfect separation just yet, but we are on it). In other words, you try to shorten the

critical rendering path that is required for the content to start displaying by pushing the content down the line as fast as possible and deferring pretty much everything else.

We followed this same separation of concerns, grouping our content types into the same categories and identifying what's critical, what's important and what's secondary. In our case, we identified and separated content in this way:

- **Core content**

Only essential HTML and CSS

- **Enhancement**

JavaScript, code syntax highlighter, full CSS, web fonts, comment ratings

- **Leftovers**

Analytics, advertising, Gravatars

Once you have this simple content/functionality priority list, improving performance is becoming just a matter of adding a few snippets for loading assets to properly reflect those priorities. Even if your server logic forces you to load all assets on all devices, by focusing on content delivery first, you ensure that the content is accessible quickly, while everything else is deferred and loaded in the background, after the page has started rendering. From a strategic perspective, the list also reflects your technical debt, as well as critical issues that slow you down. Indeed, we had quite a list of issues to deal with already at this point, so it transformed fairly quickly into a

list of content priorities. And a rather tricky issue sat right at the top of that list: good ol' web fonts.

## *Deferring Web Fonts*

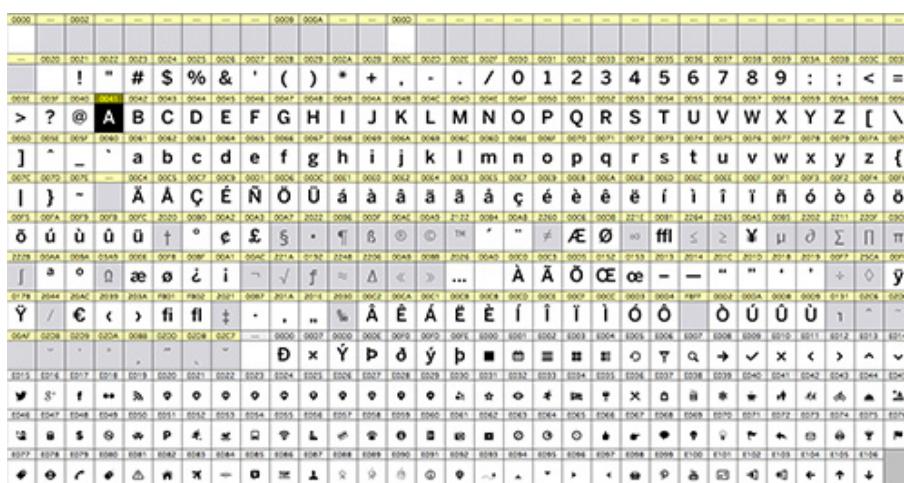
Despite the fact that the proportion of Smashing Magazine's readers on mobile has always been quite modest (just around 15%—mainly due to the length of articles), we never considered mobile as an afterthought, but we never pushed user experience on mobile either. And when we talk about user experience on mobile, we mostly talk about speed, since typography was pretty much well designed from day one.

We had conversations during the 2012 redesign about how to deal with fonts, but we couldn't find a solution that made everybody happy. The visual appearance of content was important, and because the new Smashing Magazine was all about beautiful, rich typography, not loading web fonts at all on mobile wasn't really an option.

With the redesign back then, we switched to Skolar for headings and Proxima Nova for body copy, delivered by Fontdeck. Overall, we had three fonts for each typeface — Regular, Italic and Bold — totalling in six font files to be delivered over the network. Even after our dear friends at Fontdeck subsetted and optimized the fonts, the assets were quite heavy with over 300 KB in total, and because we wanted to avoid the frequent flash of unstyled text (FOUT), we had them loaded in the header of every page. Initially we thought that the fonts would reliably be cached in HTTP cache, so they wouldn't be retrieved with every single page load. Yet it turned out that

HTTP cache was quite unreliable: the fonts showed up in the waterfall loading chart every now and again for no apparent reason, both on desktop and on mobile.

The biggest problem, of course, was that the fonts were blocking rendering<sup>32</sup>. Even if the HTML, CSS and JavaScript had already loaded completely, the content wouldn't appear until the fonts had loaded and rendered. So you had this beautiful experience of seeing link underlines first, then a few keywords in bold here and there, then subheadings in the middle of the page and then finally the rest of the page. In some cases, when Fontdeck had server issues, the content didn't appear at all, even though it was already sitting in the DOM, waiting to be displayed.



In his article, Web Fonts and the Critical Path<sup>33</sup>, Ian Feather provides a very detailed overview of the FOUT issues and font loading solutions. We tested them all.

32. <http://ianfeather.co.uk/web-fonts-and-the-critical-path/>

33. <http://ianfeather.co.uk/web-fonts-and-the-critical-path/>

We experimented with a few solutions before settling on what turned out to be perhaps the most difficult one. At first, we looked into using Typekit and Google's Web-FontLoader<sup>34</sup>, an asynchronous script which gives you more granular control of what appears on the page while the fonts are being loaded. Basically, the script adds a few classes to the **body** element, which allows you to specify the styling of content in CSS during the loading and after the fonts have loaded. So you can be very precise about how the content is displayed in fallback fonts first, before users see the switch from fallback fonts to web fonts.

We added fallback fonts declarations and ended up with pretty verbose CSS font stacks, using iOS fonts, Android fonts, Windows Phone fonts and good ol' web-safe fonts as fallbacks – we are still using these font stacks today. E.g. we used this cascade for the main headings (it reflects the order of popularity of mobile operating systems in our analytics):

```
h2 {  
    font-family: "Skolar Bold",  
    AvenirNext-Bold, "Avenir Bold",  
    "Roboto Slab", "Droid Serif",  
    "Segoe UI Bold",  
    Georgia, "Times New Roman", Times, serif;  
}
```

So readers would see a mobile OS font (or any other fallback font first), and it probably would be a font that they

---

<sup>34</sup>. <https://github.com/typekit/webfontloader>

are quite familiar with on their device, and then once the fonts have loaded, they would see a switch, triggered by WebFontLoader. However, we discovered that after switching to WebFontLoader, we started seeing FOUT way too often, with HTTP cache being quite unreliable again, and that permanent switch from a fallback font to the web font being quite annoying, basically ruining the reading experience.

So we looked for alternatives. One solution was to include the `@font-face` directive only on larger screens by wrapping it in a media query, thus avoiding loading web fonts on mobile devices and in legacy browsers altogether. (In fact, if you declare web fonts in a media query, they will be loaded only when the media query matches the screen size. So no performance hit there.) Obviously it helped us improve performance on mobile devices in no time, but we didn't feel right with having a "simplified" reading experience on mobile devices. So it was a no-go, too.

What else could we do? The only other option was to improve the caching of fonts. We couldn't do much with HTTP cache, but there was one option we hadn't looked into: storing fonts in `AppCache` or `localStorage`. Jake Archibald's article on the beautiful complexity of `AppCache`<sup>35</sup> led us away from `AppCache` to experiment with `localStorage`, a technique<sup>36</sup> that The Guardian's team was using at the time.

---

<sup>35</sup>. <http://alistapart.com/article/application-cache-is-a-douchebag>

<sup>36</sup>. <https://github.com/ahume/webfontjson>

Now, offline caching comes with one major requirement: you need to *have* the actual font files to be able to cache them locally in the client's browser. And you can't cache *a lot* because localStorage space is very limited<sup>37</sup>, sometimes with just 5Mb available per domain. Luckily, the Fontdeck guys were very helpful and forthcoming with our undertaking, so despite the fact that font delivery services usually require you to load files and have a synchronous or asynchronous callback to count the number of impressions, Fontdeck has been perfectly fine with us grabbing WOFF-files from Google Chrome's cache and setting up a "flat" pricing based on the number of page impressions in recent history.

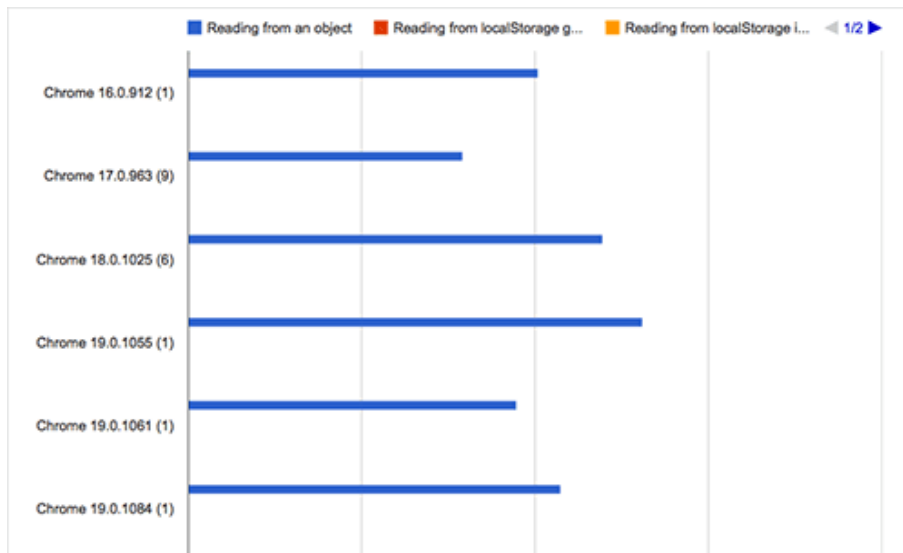
So we grabbed the WOFF files and embedded them, base64-encoded, in a single CSS file, moving from six external HTTP-requests with about 50 KB file each to at most one HTTP request on the first load and 400 KB of CSS. Obviously, we didn't want this file to be loaded on every visit. So if localStorage is available on the user's machine, we store the entire CSS file in localStorage, set a cookie and switch from the fallback font to the web font. This switch usually happens once at most because for the consequent visits, we check whether the cookie has been set and, if so, retrieve the fonts from localStorage (causing about 50ms in latency) and display the content in the web font right away. Just before you ask: yes, read/write to localStorage is much slower than retrieving files from

---

<sup>37</sup>. <http://www.html5rocks.com/en/tutorials/offline/quota-research/>



HTTP cache<sup>38</sup>, but it proved to be a bit more reliable in our case.



Yes, localStorage is much slower than HTTP cache<sup>39</sup>, but it's more reliable. Storing fonts in localStorage isn't the perfect solution, but it helped us improve performance dramatically.

If the browser doesn't support localStorage, we include fonts with good ol' **link href** and, well, frankly just hope for the best – that the fonts will be properly cached and persist in the user's browser cache. For browsers that don't support WOFF<sup>40</sup> (IE8, Opera Mini, Android <= 4.3), we provide external URLs to fonts with older font mime types, hosted on Fontdeck.

Now, if localStorage is available, we still don't want it to be blocking the rendering of the content. And we don't want to see FOUT every single time a user loads the page.

---

<sup>38</sup>. <https://github.com/addyosmani/basket.js/issues/24>

<sup>39</sup>. <https://github.com/addyosmani/basket.js/issues/24>

<sup>40</sup>. <http://caniuse.com/#search=woff>

That's why we have a little JavaScript snippet in the header before the **body** element: it checks whether a cookie has been set and, if not, we load web fonts asynchronously after the page has started rendering. Of course, we could have avoided the switch by just storing the fonts in localStorage on the first visit and have no switch during the first visit, but we decided that one switch is acceptable, because our typography is important to our identity.

The script was written, tested and documented by our good friend [Horia Dragomir](#)<sup>41</sup>. Of course, it's [available as a gist on GitHub](#)<sup>42</sup>:

```
<script type="text/javascript">
  (function () {
    "use strict";

    // once cached, the css file is stored on the client
    // forever unless the URL below is changed. Any change
    // will invalidate the cache
    var css_href = './web-fonts.css';
    // a simple event handler wrapper
    function on(el, ev, callback) {
      if (el.addEventListener) {
        el.addEventListener(ev, callback, false);
      } else if (el.attachEvent) {
        el.attachEvent("on" + ev, callback);
      }
    }
  })
}
```

---

<sup>41</sup>. <https://twitter.com/hdragomir>

<sup>42</sup>. <https://gist.github.com/hdragomir/8f00ce2581795fd7b1b7>

```

// if we have the fonts in localStorage or if we've
// cached them using the native browser cache
if ((window.localStorage &&
localStorage.font_css_cache) ||
document.cookie.indexOf('font_css_cache') > -1){
    // just use the cached version
    injectFontsStylesheet();
} else {
    // otherwise, don't block the loading of the page;
    // wait until it's done.
    on(window, "load", injectFontsStylesheet);
}

// quick way to determine whether a css file has
// been cached locally
function fileIsCached(href) {
    return window.localStorage &&
    localStorage.font_css_cache &&
    (localStorage.font_css_cache_file === href);
}

// time to get the actual css file
function injectFontsStylesheet() {
    // if this is an older browser
    if (!window.localStorage ||
!window.XMLHttpRequest) {
        var stylesheet = document.createElement('link');
        stylesheet.href = css_href;
        stylesheet.rel = 'stylesheet';
        stylesheet.type = 'text/css';
    }
}

```

```

document.getElementsByTagName('head')[0]
.appendChild(stylesheet);

// just use the native browser cache
// this requires a good expires header on the
// server
document.cookie = "font_css_cache";

// if this isn't an old browser
} else {
    // use the cached version if we already have it
    if (fileIsCached(css_href)) {
        injectRawStyle(localStorage.font_css_cache);
        // otherwise, load it with ajax
    } else {
        var xhr = new XMLHttpRequest();
        xhr.open("GET", css_href, true);
        on(xhr, 'load', function () {
            if (xhr.readyState === 4) {
                // once we have the content, quickly inject
                // the css rules
                injectRawStyle(xhr.responseText);
                // and cache the text content for further use
                // notice that this overwrites anything that
                // might have already been previously cached
                localStorage.font_css_cache =
                    xhr.responseText;
                localStorage.font_css_cache_file = css_href;
            }
        });
        xhr.send();
    }
}

```

```

    }
  }
}

// this is the simple utility that injects the cached
// or loaded css text
function injectRawStyle(text) {
  var style = document.createElement('style');
  style.innerHTML = text;
  document.getElementsByTagName('head')[0]
    .appendChild(style);
}

}());
</script>

```

During the testing of the technique, we discovered a few surprising problems. Because the cache isn't persistent in WebViews, fonts do load asynchronously in applications such as Tweetdeck and Facebook, yet they don't remain in the cache once the window is closed. In other words, with every WebViews visit, the fonts are re-downloaded. Some old Blackberry devices seemed to clear cookies and delete the cache when the battery is running out. And depending on the configuration of the device, sometimes fonts do not persist in mobile Safari either.

Still, once the snippet was in place, articles started rendering much faster. By deferring the loading of Web fonts and storing them in localStorage, we've avoided around 700ms delay, and thus shortened the critical path significantly by avoiding the latency for retrieving all the

fonts. The result was quite impressive for the first load of an uncached page, and it was even more impressive for concurrent visits since we were able to reduce the latency caused by Web fonts to just 40 to 50 ms. In fact, if we had to mention just one improvement to performance on the website, deferring web fonts is by far the most effective.

At this point, we haven't even considered using the new WOFF2 format<sup>43</sup> for fonts just yet. Currently supported in Chrome and Opera, it promises a better compression for font files and it already showed remarkable results. In fact, The Guardian was able to cut down on 200ms latency and 50 KB of the file weight<sup>44</sup> by switching to WOFF2, and we intend to look into moving to WOFF2 soon as well.

Of course, grabbing WOFFs might not always be an option for you, but it wouldn't hurt just to talk to type foundries to see where you stand or to work out a deal to host fonts "locally." Otherwise, tweaking WebFontLoader for Typekit and Fontdeck is definitely worth considering.

## *Dealing With JavaScript*

With the goal of removing all unnecessary assets from the critical rendering path, the second target we decided to deal with was JavaScript. And it's not like we particularly dislike JavaScript for some reason, but we always tend to prefer non-JavaScript solutions to JavaScript ones.

---

<sup>43</sup>. <https://gist.github.com/sergejmueller/cf6b4f2133bcb3e2f64a>

<sup>44</sup>. <https://twitter.com/patrickhamann/status/49776778703933442>

In fact, if we can avoid JavaScript or replace it with CSS, then we'll always explore that option.

Back in 2012, we weren't using a lot of scripts on the page, yet displaying advertising via OpenX depended on jQuery, which made it way too easy to lazily approach simple, straightforward tasks with ready-to-use jQuery plugins. At the time, we also used Respond.js to emulate responsive behaviour in legacy browsers. However, Internet Explorer 8 usage has dropped significantly between 2012 and 2014: with 4.7% before the redesign, it was now 1.43%, with a dropping tendency every single month. So we decided to deliver a fixed-width layout with a specific IE8.css stylesheet to those users, and removed Respond.js altogether.

As a strategic decision, we decided to defer the loading of all JavaScripts until the page has started rendering and we looked into replacing jQuery with lightweight modular JavaScript components.

jQuery was tightly bound to ads, and ads were supposed to start displaying as fast as possible, so to make it happen, we had to deal with advertising first. The decision to defer the loading of ads wasn't easy to get agreement on, but we managed to make a convincing argument that better performance would increase click rates because users would see the content sooner. That is, on every page, readers would be attracted by the high-quality content and then, when the ads kick in, would pay attention to those squares in the sidebar as well.

Florian Sander<sup>45</sup>, our partner in crime when it comes to advertising, rewrote the script for our banner ads so that banners would be loaded only after the content has started rendering, and only then the advertising spots would be put into place. Florian was able to get rid of two render-blocking HTTP-requests that the ad-script normally generated, and we were able to remove the dependency on jQuery by rewriting the script in vanilla JavaScript.

Obviously, because the sidebar's ad content is generated on the fly and is loaded after the render tree has been constructed, we started seeing reflows (this still happens when the page is being constructed). Because we used to load ads before the content, the entire page (with pretty much everything) used to load at once. Now, we've moved to a more modular structure, grouping together particular parts of the page and queuing them to load after each other. Obviously, this has made the overall experience on the site a bit noisier because there are a few jumps here and there, in the sidebar, in the comments and in the footer. That was a compromise we went for, and we are working on a solution to reserve space for "jumping" elements to avoid reflows as the page is being loaded.

## DEFERRING NON-CRITICAL JAVASCRIPT

When the prospect of removing jQuery altogether became tangible as a long-term goal, we started working step by step to decouple jQuery dependencies from the li-

---

<sup>45</sup>. <http://www.kreativrauschen.de/>



brary. We rewrote the script to generate footnotes for the print style sheet (later replacing it with a PHP solution), rewrote the functionality for rating comments, and rewrote a few other scripts. Actually, with our savvy user base and a solid share of smart browsers, we were able to move to vanilla JavaScript quite quickly. Moreover, we could now move scripts from the header to the footer to avoid blocking construction of the DOM tree. In mid-July, we removed jQuery from our code base entirely.

We wanted full control of what is loaded on the page and when. Specifically, we wanted to ensure that no JavaScript blocks the rendering of content at any point. So, we use the Defer Loading JavaScript<sup>46</sup> script to load JavaScript after the `load` event by injecting the JavaScript after the DOM and CSSOM have already been constructed and the page has been painted. Here's the snippet that we use on the website, with the `defer.js` script (which is loaded asynchronously after the `load` event):

```
function downloadJSAtOnload() {  
    var element = document.createElement("script");  
    element.src = "defer.js";  
    document.body.appendChild(element);  
}  
  
if (window.addEventListener)  
    window.addEventListener("load", downloadJSAtOnload,  
        false);  
else if (window.attachEvent)
```

---

<sup>46</sup>. <http://www.feedthebot.com/pagespeed/defer-loading-javascript.html>

```
window.attachEvent("onload", downloadJSAtOnload);  
else  
    window.onload = downloadJSAtOnload;
```

However, because script-injected asynchronous scripts are considered harmful<sup>47</sup> and slow (they block the browser's speculative parser), we might be looking into using the good ol' **defer** and **async** attributes instead. In the past, we couldn't use **async** for every script because we needed jQuery to load before its dependencies; so, we used **defer**, which respects the loading order of scripts. With jQuery out of the picture, we can now load scripts asynchronously, and fast. Actually by the time you read this chapter, we might already be using **async**.

Basically, we just deferred the loading of all JavaScripts that we identified previously, such as syntax highlighter and comment ratings, and cleared a path in the header for HTML and CSS.

## *Inlining Critical CSS*

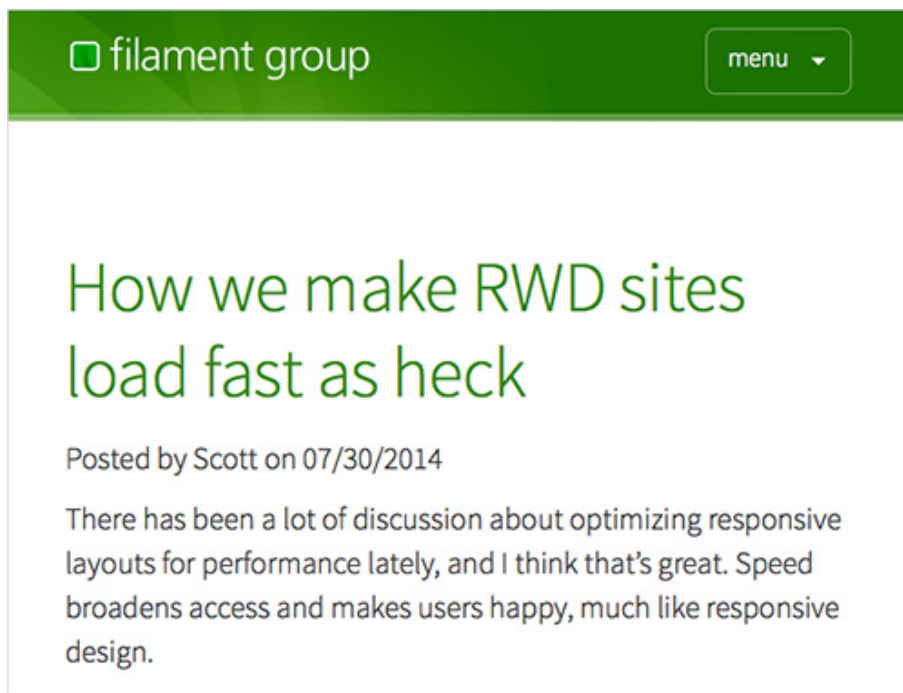
That wasn't good enough, though. Performance did improve dramatically; however, even with all of these optimizations in place, we didn't hit that magical Speed Index value of under 1000. In light of the ongoing discussion about inline CSS and above-the-fold CSS, as recommended by Google<sup>48</sup>, we looked into more radical ways to deliv-

---

<sup>47</sup>. <https://www.igvita.com/2014/05/20/script-injected-async-scripts-considered-harmful/>

<sup>48</sup>. <https://developers.google.com/web/fundamentals/performance/critical-rendering-path/page-speed-rules-and-recommendations>

er content quickly. To avoid an HTTP request when loading CSS, we measured how fast the website would be if we were to load critical CSS inline and then load the rest of the CSS once the page has rendered.



Scott Jehl's [article](http://www.filamentgroup.com/lab/performance-rwd.html)<sup>49</sup> explains how exactly to extract and inline critical CSS.

But what exactly is critical CSS? And how do you extract it from a potentially complex code base? As [Scott Jehl points out](http://www.filamentgroup.com/lab/performance-rwd.html)<sup>50</sup>, critical CSS is the subset of CSS that is needed to render the top portion of the page across all breakpoints. What does that mean? Well, you would decide on a certain height that you would consider to be “above the fold” content — it could be 600, 800 or 1200 pixels or anything else — and you would collect into their own style

---

<sup>49</sup>. <http://www.filamentgroup.com/lab/performance-rwd.html>

<sup>50</sup>. <http://www.filamentgroup.com/lab/performance-rwd.html>

sheet all of the styles that specify how to render content within that height across all screen widths.

Then you inline those styles in the **head**, and thus give the browser everything it needs to start render that visible portion of the page – within one single HTTP request. You've heard it a few times by now: everything else is deferred after the first initial rendering. You avoid an HTTP-request, and you load the full CSS asynchronously, so once the user starts scrolling, the full CSS will (hopefully) already have loaded.

Visually speaking, content will appear to render more quickly, but there will also be more reflowing and jumping on the page. So, if a user has followed a link to a particular comment below the “fold”, then they will see a few reflows as the website is being constructed because the page is rendered with critical CSS first (there is just so much we can fit within 14 KB!) and adjusted later with the complete CSS. Of course, inline CSS isn't cached; so, if you have critical CSS and load the complete CSS on rendering, it's useful to set a cookie, so that inline styles aren't inlined with every single load. The drawback of course is that you might have duplicate CSS because you would be defining styles both inline and in the full CSS, unless you're able to strictly separate them.

Because we had just refactored our CSS code base, identifying critical CSS wasn't very difficult. Obviously, there are smart<sup>51</sup> tools<sup>52</sup> that analyze the markup and CSS, identify critical CSS styles and export them into a sepa-

---

<sup>51</sup>. <http://css-tricks.com/authoring-critical-fold-css/>

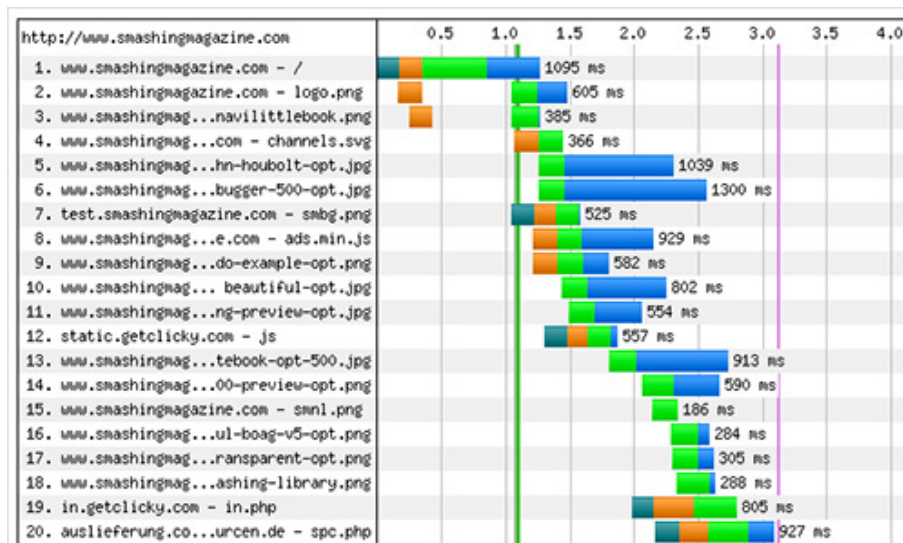
<sup>52</sup>. <https://github.com/addyosmani/above-the-fold-css-tools>

rate file during the build process, but we were able to do it manually. Again, you have to keep in mind that 14 Kb is your budget for HTML and CSS, so in the end we had to rename a few classes here and there, and compress CSS as well.

We analyzed the first 800px, checking the inspector for the CSS that was needed and separating our style sheet into two files — and actually that was pretty much it. One of those files, *above-the-fold.css*, is minified and compressed, and its content is placed inline in the head of our document as early as possible — not blocking rendering. The other file, our full CSS file, is then loaded with JavaScript after the content has loaded, and if JavaScript isn't available for some reason or the user is on a legacy browser, we've put a full CSS file inside **noscript** tag at the end of the head, so they don't get an unstyled HTML page.

## *Was It All Worth It?*

Because we've just implemented these optimizations, we haven't been able to measure their impact on traffic, but we'll publish these results later as well. Obviously, we *did* notice a quite remarkable technical improvement though. By deferring and caching web fonts, inlining CSS and optimizing the critical rendering path for the first 14Kb, we were able to achieve dramatic improvements in loading times. The start rendering time started circling around 1s for an uncached page on 3G and was around 700ms (including latency!) on subsequent loads.



We've been using [WebPageTest](http://www.webpagetest.org/)<sup>53</sup> a lot for running tests. Our waterfall chart was becoming better over time and reflected the priorities we had defined earlier. [Large view](http://www.webpagetest.org/result/140904_H4_T5R/1/details/).<sup>54</sup>

On average, Smashing Magazine's front page makes 45 HTTP-requests and has 440 KB in bandwidth on the first uncached load. Because we heavily cache everything but ads, subsequent visits have around 15 HTTP requests and 180 KB of traffic. The First Byte time is still around 300–600ms (which is *a lot*), yet Start Render time is usually under 0.7s<sup>55</sup> on a DSL connection in Amsterdam (for the very first, uncached load), and usually under 1.7s on a slow 3G<sup>56</sup>. On a fast cable connection, the site starts rendering within 0.8s<sup>57</sup>, and on a fast 3G, within 1.1s<sup>58</sup>. Obviously, the results vary significantly depending on the

<sup>53</sup>. <http://www.webpagetest.org/>

<sup>54</sup>. [http://www.webpagetest.org/result/140904\\_H4\\_T5R/1/details/](http://www.webpagetest.org/result/140904_H4_T5R/1/details/)

<sup>55</sup>. [http://www.webpagetest.org/result/140904\\_ZJ\\_T62/](http://www.webpagetest.org/result/140904_ZJ_T62/)

<sup>56</sup>. [http://www.webpagetest.org/result/140904\\_Y5\\_SXS/](http://www.webpagetest.org/result/140904_Y5_SXS/)

<sup>57</sup>. [http://www.webpagetest.org/result/140904\\_DB\\_T5Y/](http://www.webpagetest.org/result/140904_DB_T5Y/)

<sup>58</sup>. [http://www.webpagetest.org/result/140904\\_H4\\_T5R/](http://www.webpagetest.org/result/140904_H4_T5R/)

First Byte time which we can't improve just yet, at the time of writing. That's the only asset that introduces unpredictability into the loading process, and as such has a decisive impact on the overall performance.

Just by following basic guidelines by our colleagues mentioned above and Google's recommendations, we were able to achieve the 97–99 Google PageSpeed score<sup>59</sup> both on desktop and on mobile. The score varies depending on the quality and the optimization level of advertising assets displayed randomly in the sidebar. Again, the main culprit is the server's response time — not for long, though.

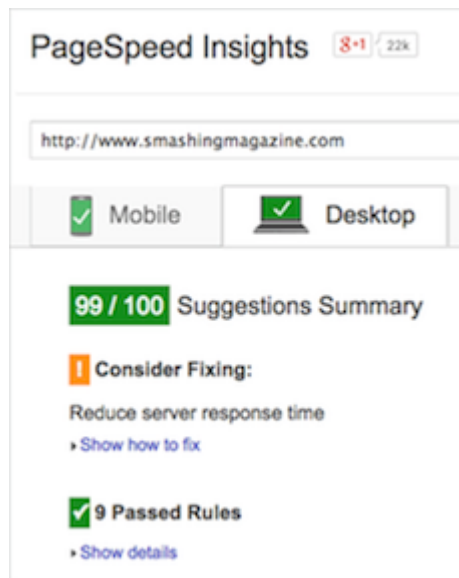
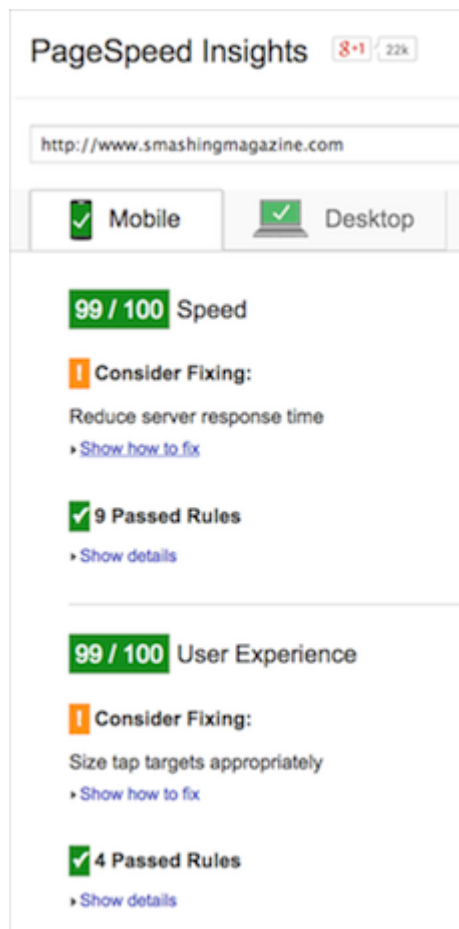
By the way, Scott Jehl has also published a wonderful article on the front-end techniques<sup>60</sup> FilamentGroup uses to extract critical CSS and load it inline while loading the full CSS afterwards and avoid downloading overheads. Patrick Hamann's talk on “Breaking News at 1000ms”<sup>61</sup> explains a few techniques that The Guardian is using to hit the SpeedIndex 1000 mark. Definitely worth reading and watching, and indeed quite similar to what we implemented on this very site as well.

---

<sup>59.</sup> <https://developers.google.com/speed/pagespeed/insights/?url=http%3A%2F%2Fwww.smashingmagazine.com&tab=desktop>

<sup>60.</sup> <http://filamentgroup.com/lab/performance-rwd.html>

<sup>61.</sup> <https://www.youtube.com/watch?v=dfweWyVScaI>



After a few optimizations, we achieved a Google PageSpeed score of 99 on mobile<sup>62</sup>. We got a Google PageSpeed score of 99 on the desktop<sup>63</sup> as well.



## Work To Be Done

While the results we were able to achieve are quite satisfactory, there is still a lot of work to be done. For example, we haven't considered optimizing the delivery of images just yet, and are now adjusting our editorial process to integrate the new `picture` element and `srcset/sizes` with [Picturefill 2.1.0](http://scottjehl.github.io/picturefill/)<sup>64</sup>, to make the loading even faster on mobile devices. At the moment, all images have a fixed width of 500px and are basically scaled down on smaller views. Every image is optimized and compressed, but we don't deliver different images for different devices — and no, we aren't delivering any Retina images at all. That is all about to change soon.

While Smashing Magazine's home page is well optimized, some pages and articles still perform poorly. Articles with many comments are quite slow because we use [Gravatar.com](https://en.gravatar.com/)<sup>65</sup> for comments. Because each Gravatar URL is unique, each comment generates one HTTP request, slowing down the loading of the overall page. We are going to defer the loading of Gravatars and cache them locally with a Gravatar Cache WordPress plugin<sup>66</sup>. We might have already done it by the time you read this.

We're playing around with DNS prefetching and HTML5 preloading to resolve DNS lookups way ahead of

---

<sup>62</sup>. [https://developers.google.com/speed/pagespeed/insights/](https://developers.google.com/speed/pagespeed/insights/?url=http%3A%2F%2Fwww.smashingmagazine.com&tab=mobile)

[?url=http%3A%2F%2Fwww.smashingmagazine.com&tab=mobile](https://developers.google.com/speed/pagespeed/insights/?url=http%3A%2F%2Fwww.smashingmagazine.com&tab=mobile)

<sup>63</sup>. [https://developers.google.com/speed/pagespeed/insights/](https://developers.google.com/speed/pagespeed/insights/?url=http%3A%2F%2Fwww.smashingmagazine.com&tab=desktop)

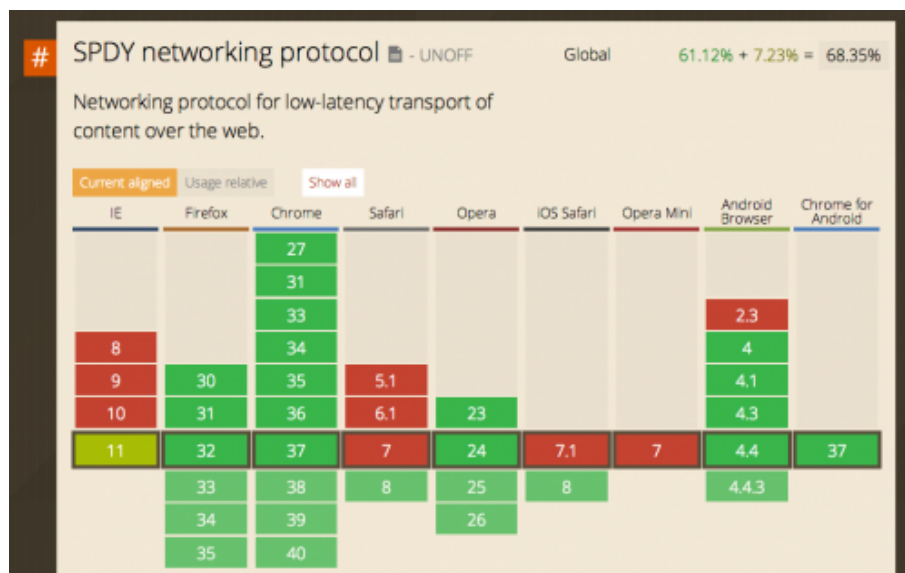
[?url=http%3A%2F%2Fwww.smashingmagazine.com&tab=desktop](https://developers.google.com/speed/pagespeed/insights/?url=http%3A%2F%2Fwww.smashingmagazine.com&tab=desktop)

<sup>64</sup>. <http://scottjehl.github.io/picturefill/>

<sup>65</sup>. <https://en.gravatar.com/>

<sup>66</sup>. <https://wordpress.org/plugins/fv-gravatar-cache/>

time (for example, for Gravatars and advertising). However, we are being careful and hesitant here because we don't want to create a loading overhead for users on slow or expensive connections. Besides, we've added third-party meta data<sup>67</sup> to make our articles a bit easier to share. So, if you link to an article on Facebook, Facebook will pull optimized images, a description and a title from our meta data, which is crafted individually for each article. We've also happily noticed that article pages scroll smoothly at 60fps<sup>68</sup>, and that with relatively large images and ads.



Yes, we can use SPDY today<sup>69</sup>. We just need to install SPDY Nginx Module<sup>70</sup> or Apache SPDY Module<sup>71</sup>. This is what we are going to tackle next.

<sup>67</sup>. <http://alistapart.com/article/like-able-content-spread-your-message-with-third-party-metadata>

<sup>68</sup>. <http://jankfree.org>

<sup>69</sup>. <http://caniuse.com/#search=SPDY>

<sup>70</sup>. [http://nginx.org/en/docs/http/nginx\\_http\\_spdy\\_module.html](http://nginx.org/en/docs/http/nginx_http_spdy_module.html)

<sup>71</sup>. <https://code.google.com/p/mod-spdy/>

Despite all of our optimizations, the main issue still hasn't been resolved: very slow servers and the First Byte response times. We've been experiencing difficulties with our current server setup and architecture but are tied with a long-term contract, yet we will be moving to a new server soon. We'll take that opportunity to also move to SPDY<sup>72</sup> on the server, a predecessor of HTTP 2.0 (which is well supported in major browsers<sup>73</sup>, by the way), and we are looking into using a content delivery network as well.

## PERFORMANCE OPTIMIZATION STRATEGY

To sum up, optimizing the performance of Smashing Magazine was quite an effort to figure out, yet many aspects of optimization can be achieved very quickly. In particular, front-end optimization is quite easy and straightforward as long as you have a shared understanding of priorities. Yes, that's right: you optimize content delivery, and defer everything else.

Strategically speaking, the following could be your performance optimization roadmap:

- Remove blocking scripts from the header of the page.
- Identify and defer non-critical CSS and JavaScript.
- Identify critical CSS and load it inline in the head, and then load the full CSS after rendering. (Make sure to set a

---

<sup>72</sup>. <https://developers.google.com/speed/spdy/>

<sup>73</sup>. <http://caniuse.com/#search=SPDY>

cookie to prevent inline styles from loading with every page load.)

- Keep all critical HTML and CSS to under 14 KB, and aim for a Speed Index of under 1000.
- Defer the loading of Web fonts and store them in local-Storage or AppCache.
- Consider using WOFF2 to further reduce latency and file size of the web fonts.
- Replace JavaScript libraries with leaner JavaScript modules.
- Avoid unnecessary libraries, and look into options for removing Respond.js and Modernizr; for example, by “[cutting the mustard](#)<sup>74</sup>” to separate browsers into buckets. Legacy browsers could get a fixed-width layout. [Clever SVG fallbacks](#)<sup>75</sup> also exist.

That’s basically it. By following these guidelines, you can make your responsive website really, *really* fast.

## Conclusion

Yes, finding just the right strategy to make this very website fast took a lot of experimentation, blood, sweat and cursing. Our discussions kept circling around next steps and on critical and not-so-critical components and some-

---

<sup>74</sup>. <http://responsivenews.co.uk/post/18948466399/cutting-the-mustard>

<sup>75</sup>. <http://css-tricks.com/svg-fallbacks/>

times we had to take three steps back in order to pivot in a different direction. But we learned a lot along the way, and we have a pretty clear idea of where we are heading now, and, most importantly, how to get there.

So here you have it. A little story about the things that happened on this little website over the last year. If you notice any issues, please let us know on Twitter [@smashingmag](https://twitter.com/smashingmag)<sup>76</sup> and we'll hunt them down for good.

Ah, and thanks for keeping us reading throughout all these years. It means *a lot*. You are quite smashing indeed. You should know that. 🐼

*A big “thank you” to Patrick Hamann and Jake Archibald for the technical review of the article as well as Andy Hume and Tim Kadlec for their fantastic support throughout the years. Also a big “thank you” to our front-end engineer, Marco, for his help with the article and for his thorough and tireless front-end work, which involved many experiments, failures and successes along the way. Also, kind thanks to the Inpsyde team and Florian Sander for technical implementations.*

*A final thank you goes out to Iris, Melanie, Cosima and Markus for keeping an eye out for those nasty bugs and looking after the content on the website. Without you, this website wouldn't exist. And thank you for having my back all this time. I respect and value every single bit of it. You rock.*

---

<sup>76</sup>. <http://www.twitter.com/smashingmag>

# How To Speed Up Your WordPress Website

BY MARCUS TAYLOR 🍷

A few months ago, I ran an experiment to see how much faster I could make one of my websites in less than two hours of work. After installing a handful of WordPress plugins and fixing a few simple errors, I had improved the website's loading speed from 1.61 seconds to 583 milliseconds. That's a 70.39% improvement, without having made any visual changes to the website.

According to a 2009 Akamai study<sup>77</sup>, 47% of visitors expect a page to load in under 2 seconds, and 57% of visitors will abandon a page that takes more than 3 seconds to load. Since this study, no shortage of case studies have confirmed that loading time affects sales.

In 2006, Amazon reported that a 100-millisecond increase in page speed translated to a 1% increase in its revenue. Just a few years later, Google announced in a blog post<sup>78</sup> that its algorithm takes page speed into account when ranking websites.

---

<sup>77</sup>. [http://www.akamai.com/html/about/press/releases/2009/press\\_091409.html](http://www.akamai.com/html/about/press/releases/2009/press_091409.html)

<sup>78</sup>. <http://googlewebmastercentral.blogspot.com.au/2010/04/using-site-speed-in-web-search-ranking.html>



*So, how can you speed up your WordPress website?*

Below are twelve quick fixes that will dramatically improve your website's loading time, including:

- identifying which plugins are slowing down your website;
- automatically compressing Web pages, images, JavaScript and CSS files;
- keeping your website's database clean;
- setting up browser caching the right way.

## *Lay The Foundation*

When your house is sinking into the ground, you don't polish the windows — you fix the foundations. The same goes for your website. If it's hosted on a sluggish server or has a bloated theme, quick fixes won't help. You'll need to fix the foundation.

So, let's start with what makes for a good foundation and how to set ourselves up for a website that runs at lightening speed.

## CHOOSE A GOOD HOST

Your Web hosting company and hosting package have a huge impact on the speed of your website, among many other important performance-related things. I used to be sucked in by the allure of free or cheap hosting, but with the wisdom of hindsight, I've learned that hosting isn't an area to skimp on.

To put this into perspective, two of my clients have similar websites but very different hosting providers. One uses WPEngine (an excellent hosting company), and the other hosts their website on a cheap shared server.

The DNS response time (i.e. the time it takes for the browser to connect to the hosting server) of the client using WPEngine is 7 milliseconds. The client using the cheap shared hosting has a DNS response time of 250 milliseconds.

If you want your website to run quickly, start with a good hosting company<sup>79</sup> and package.

## CHOOSE A GOOD THEME

Unfortunately, not all WordPress themes are created equal. While some are extremely fast and well coded, others are bloated with hundreds of bells and whistles under the pretence of being “versatile and customizable.”

---

<sup>79</sup>. <http://www.ventureharbour.com/web-hosting-guide/>



A few years ago, Julian Fernandes of Synthesis ran an interesting case study in which he updated his theme from WordPress' default to the Genesis framework, monitoring page speed. He noticed that just by changing the theme to Genesis, his loading time improved from 630 to 172 milliseconds.

When you choose a theme, check the page speed of the theme's demo, using a tool such as Pingdom, to see how quickly it runs with nothing added to it. This should give you an idea of how well coded it is.

## USE A CONTENT DELIVERY NETWORK

I recently started using a content delivery network (CDN) for one of my websites and noticed a 55% reduction in bandwidth usage and a huge improvement in page-loading speed.

A CDN hosts your files across a huge network of servers around the world. If a user from Argentina visits your website, then they would download files from the server closest to them geographically. Because your bandwidth is spread across so many different servers, the load on any single server is reduced.

Setting up a CDN can take a few hours, but it's usually one of the quickest ways to dramatically improve page-loading speed.

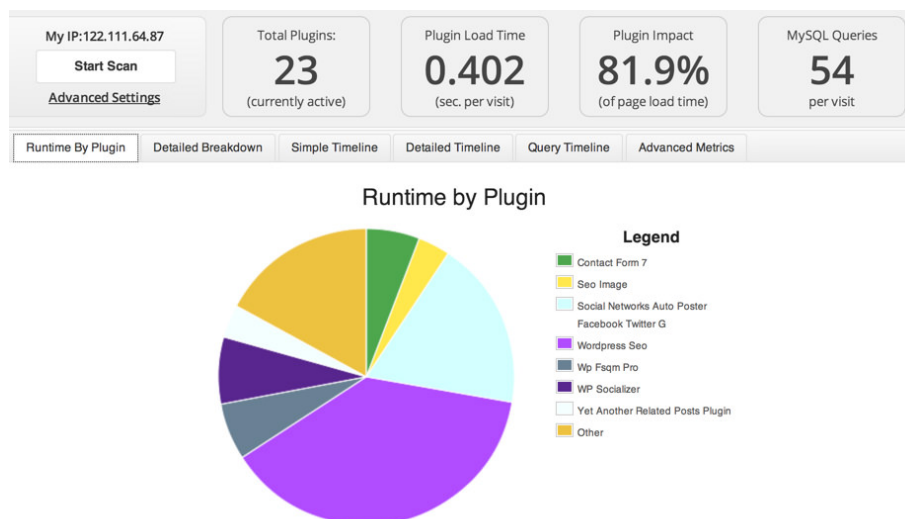
## *12 Quick Fixes To Speed Up WordPress*

Now that our foundation is solid, we can begin fine-tuning our website.

A good way to start speeding up a website is to look at what can be removed. More often than not, a website is slow not because of what it lacks but because of what it already has.

## 1. IDENTIFY PLUGINS THAT ARE SLOWING YOU DOWN

P3<sup>80</sup> is one of my favourite diagnostic plugins because it shows you the impact of your other plugins on page-loading time. This makes it easy to spot any plugins that are slowing down your website.



A common culprit is social-sharing plugins, most of which bloat page-loading times and can easily be replaced by embedding social buttons into the theme's source code.

Once you're aware of which plugins are slowing down your website, you can make an informed decision about

---

<sup>80</sup>. <https://wordpress.org/plugins/p3-profiler/>

whether to keep them, replace them or remove them entirely.

## 2. COMPRESS YOUR WEBSITE

When you compress a file on your computer as a ZIP file, the total size of the file is reduced, making it both easier and faster to send to someone. Gzip works in exactly the same way but with your Web page files.

Once installed, Gzip automatically compresses your website's files as ZIP files, saving bandwidth and speeding up page-loading times. When a user visits your website, their browser will automatically unzip the files and show their contents. This method of transmitting content from the server to the browser is far more efficient and saves a lot of time.


### Check GZIP compression

With this tool you can check if your webserver is sending the GZIP compressed header to your clients. By enabling GZIP compression on your server, you can save around 50% on your bandwidth usage.

Website URL

Check

#### Results for <http://www.musiclawcontracts.com>

**YES, it's GZIP enabled!**

Uncompressed size:	68,281 bytes
Compressed size:	13,167 bytes

By compressing this page with GZIP, **80.7% bandwidth was saved.**

There is virtually no downside to installing Gzip, and the increase in speed can be quite dramatic. As we can see in

the screenshot above, [MusicLawContracts.com](http://www.musiclawcontracts.com)<sup>81</sup> goes from 68 KB to only 13 KB with Gzip installed.

While some plugins will add Gzip to your website with the click of a button, installing it manually is actually very simple. Open your `.htaccess` file (found in the root directory on your server), and add the following code to it:

```
AddOutputFilterByType DEFLATE text/plain
AddOutputFilterByType DEFLATE text/html
AddOutputFilterByType DEFLATE text/xml
AddOutputFilterByType DEFLATE text/css
AddOutputFilterByType DEFLATE application/xml
AddOutputFilterByType DEFLATE application/xhtml+xml
AddOutputFilterByType DEFLATE application/rss+xml
AddOutputFilterByType DEFLATE application/javascript
AddOutputFilterByType DEFLATE application/x-javascript
```

Once you've added this snippet of code to `.htaccess`, test whether Gzip is working on your website by running [Check Gzip Compression](http://checkgzipcompression.com/)<sup>82</sup>. If for whatever reason the code above doesn't work, try one of the other methods that Patrick Sexton describes in his article "[Enable Gzip](http://www.feedthebot.com/pagespeed/enable-compression.html)"<sup>83</sup>.

---

<sup>81</sup>. <http://www.musiclawcontracts.com>

<sup>82</sup>. <http://checkgzipcompression.com/>

<sup>83</sup>. <http://www.feedthebot.com/pagespeed/enable-compression.html>

### 3. COMPRESS IMAGES

Images take up the majority of bandwidth on most websites. WP Smush.it<sup>84</sup> is another great plugin that automatically compresses images as you upload them to the media library. All compression is “lossless,” meaning that you won’t notice any difference in the quality of images.

One nice thing about WP Smush.it is that it works retroactively. If thousands of images are saved in your media library, you can run them all through the plugin, compressing them to a more manageable size.

### 4. LEVERAGE BROWSER CACHING

Browser caching is a tricky issue. A handful of great caching plugins are available, but if set up incorrectly, they could cause more harm than good<sup>85</sup>.

Expires headers tell the browser whether to request a particular file from the server or from the browser’s cache. Of course, this only works if the user already has a version of your Web page stored in their cache; so, the technique will speed up the website only for those who have already visited your website.

Expires headers speed up a website in two ways. First, they reduce the need for returning visitors to download the same files from your server twice. Secondly, they reduce the number of HTTP requests made.

To do this with a plugin, I recommend using WP Super Cache<sup>86</sup>. However, following an installation guide<sup>87</sup> is

---

<sup>84</sup>. <https://wordpress.org/plugins/wp-smushit/>

<sup>85</sup>. <http://www.smashingmagazine.com/2014/03/21/wordpress-performance-improvements-that-can-go-wrong/>

strongly recommended to ensure that you set it up correctly. Alternatively, you could add expires headers by adding the following code to your `.htaccess` file.

```
#
# associate .js with "text/javascript" type (if not present
# in mime.conf)
#
AddType text/javascript .js

#
# configure mod_expires
#
# URL: http://httpd.apache.org/docs/2.2/mod/mod_expires.html
#

ExpiresActive On
ExpiresDefault "access plus 1 seconds"
ExpiresByType image/x-icon "access plus 2692000 seconds"
ExpiresByType image/jpeg "access plus 2692000 seconds"
ExpiresByType image/png "access plus 2692000 seconds"
ExpiresByType image/gif "access plus 2692000 seconds"
ExpiresByType application/x-shockwave-flash "access plus
2692000 seconds"
ExpiresByType text/css "access plus 2692000 seconds"
ExpiresByType text/javascript "access plus 2692000 seconds"
ExpiresByType application/x-javascript "access plus 2692000
```

---

86. <https://wordpress.org/plugins/wp-super-cache/>

87. <http://www.wpbeginner.com/beginners-guide/how-to-install-and-setup-wp-super-cache-for-beginners/>

```
seconds"
ExpiresByType text/html "access plus 600 seconds"
ExpiresByType application/xhtml+xml "access plus 600 seconds"

#
# configure mod_headers
#
# URL: http://httpd.apache.org/docs/2.2/mod/mod_headers.html
#

Header set Cache-Control "max-age=2692000, public"

Header set Cache-Control "max-age=600, private,
must-revalidate"

Header unset ETag
Header unset Last-Modified
```

## 5. CLEAN UP THE DATABASE

I'm a big fan of how often WordPress autosaves everything, but the disadvantage is that your database will get filled with thousands of post revisions, trackbacks, pingbacks, unapproved comments and trashed items pretty quickly.

The solution to this is a fantastic plugin called WP-Optimize<sup>88</sup>, which routinely clears out your database's trash, keeping the database efficient and filled only with what

---

<sup>88</sup>. <https://wordpress.org/plugins/wp-optimize/>

needs to be kept. Of course, when doing anything to your database, always back up first.

## 6. MINIFY CSS AND JAVASCRIPT FILES

If you've installed more than a handful of plugins, chances are that your website links to 10 to 20 individual style sheets and JavaScript files on every page. This is not ideal. Putting all JavaScript into one JavaScript file and all CSS in one CSS file is considerably more efficient.

This is where minification comes in. Plugins such as Better WordPress Minify<sup>89</sup> will combine all of your style sheets and JavaScript files into one, reducing the number of requests that the browser needs to make.

I prefer Better WordPress Minify because it's less aggressive than some of the other plugins that do the same thing (some of which cause problems, as Hristo Pandjarov outlines<sup>90</sup>).

## 7. TURN OFF PINGBACKS AND TRACKBACKS

Pingbacks and trackbacks are methods used by WordPress to alert other blogs that your posts link to. While sometimes interesting, they can be a drain on page speed and are usually better turned off. You can turn them off under the "Discussion" tab in "Settings."

---

89. <https://wordpress.org/plugins/bwp-minify/>

90. <http://www.smashingmagazine.com/2014/03/21/wordpress-performance-improvements-that-can-go-wrong/>



## 8. SPECIFY IMAGE DIMENSIONS AND CHARACTER SETS

Before a visitor's browser can display your Web page, it has to figure out how to lay out the content around the images. Without knowing the size of these images, the browser has to figure it out, causing it to work harder and take longer.

Specifying image dimensions saves the browser from having to go through this step, speeding things up.

For the same reason, specifying a character set in your HTTP response headers is useful, so that the browser doesn't have to spend extra time working out which one you're using. Simply add the character set to your website's **head** section.

## 9. MOVE CSS TO THE TOP AND JAVASCRIPT TO THE BOTTOM

Linking to your style sheets as close to the top of the page as possible is widely recommended because browsers won't render a page before rendering the CSS file.

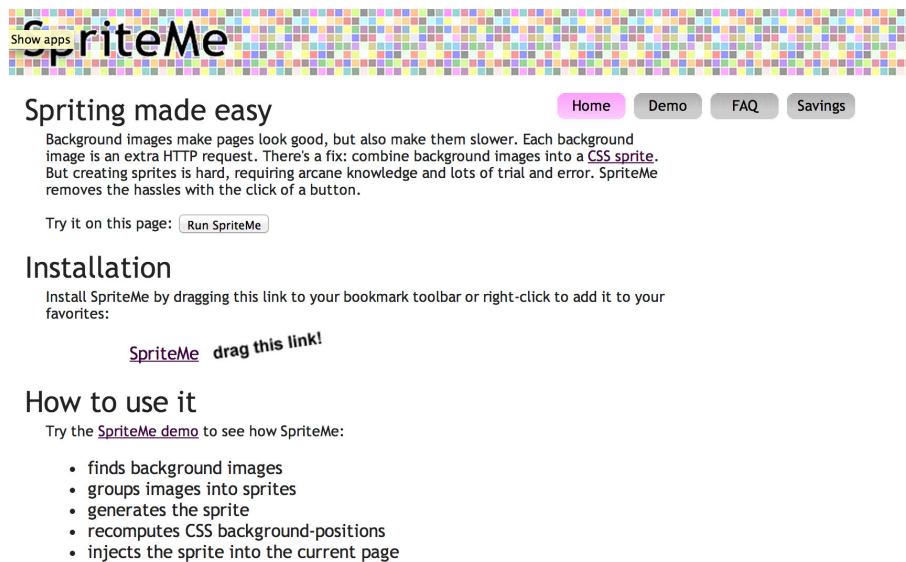
JavaScript, on the other hand, should be as close to the bottom of the footer as possible because it prevents browsers from parsing anything after it until it has full loaded.

In the majority of cases, this simple fix will improve page-loading speed by forcing files to be downloaded in the optimal order. But it can cause issues on websites that rely heavily on JavaScript and that require JavaScript files to load before the user sees any of the page.

## 10. USE CSS SPRITES

A sprite is essentially one large image file that contains all of your individual images next to each other. Using CSS, you can hide everything in the image except for the section you need, by specifying a set of coordinates.

CSS sprites speed up a website because loading one big image is much faster than loading a lot of small images.



The easiest solution is [SpriteMe](http://spriteme.org/)<sup>91</sup>, a tool that turns all of your images into a CSS sprite.

Remember that Safari does not load large sprites, so use [William Malone's calculator](http://www.williammalone.com/articles/html5-javascript-ios-maximum-sprite-frames/)<sup>92</sup> to identify whether your sprite is too large.

---

<sup>91</sup>. <http://spriteme.org/>

<sup>92</sup>. <http://www.williammalone.com/articles/html5-javascript-ios-maximum-sprite-frames/>

## 11. ENABLE KEEP ALIVE

HTTP Keep Alive refers to the message that is sent between the client's machine and the Web server asking for permission to download a file. Enabling Keep Alive allows the client's machine to download multiple files without having to repeatedly ask for permission, thus saving bandwidth.

To enable Keep Alive, simply copy and paste the code below into your `.htaccess` file.

```
Header set Connection keep-alive
```

## 12. REPLACE PHP WITH STATIC HTML WHERE APPROPRIATE

PHP is great for making a website efficient and reducing the need to enter the same information multiple times. However, calling information through PHP uses up server resources and should be replaced with static HTML where it doesn't save any time.

## Conclusion

In the next 12 months, mobile Internet usage is expected to overtake desktop usage. This shift towards Internet-enabled mobile devices means that having a fast website has never been as important as it is today. Users now expect websites to be lightening fast, and developers who don't comply will ultimately lose out to developers who invest in delivering a great experience. 🐼

# You May Be Losing Users If Responsive Web Design Is Your Only Mobile Strategy

MAXIMILIANO FIRTMAN 🍷

You resize the browser and a smile creeps over your face. You're happy: You think you are now mobile-friendly, that you have achieved your goals for the website. Let me be a bit forward before getting into the discussion: You are losing users and probably money if responsive web design is your entire goal and your only solution for mobile. The good news is that you can do it right.

In this chapter, we'll cover the relationship between the mobile web and responsive design, starting with how to apply responsive design intelligently, why performance is so important in mobile, why responsive design should not be your website's goal, and ending with the performance issues of the technique to help us understand the problem.

Designers and developers have been oversimplifying the problem of mobile since 2000, and some people now think that responsive web design is the answer to all of our problems.

We need to understand that, beyond any other goal, a mobile web experience must be lightning fast. Delivering a fast, usable and compatible experience to all mobile devices has always been a challenge, and it's no different when you are implementing a responsive technique. Embracing performance from the beginning is easier.

Responsive web design is great, but it's not a silver bullet. If it's your only weapon for mobile, then a performance problem might be hindering your conversion rate. Around 11% of the websites are responsive<sup>93</sup>, and the number is growing every month, so now is the time to talk about this.

According to Guy Podjarny's research<sup>94</sup>, 72% of responsive websites deliver the same number of bytes regardless of screen size, even on slow mobile network connections. Not all users will wait for your website to load.

With just a basic understanding of the problem, we can minimize this loss.

## MOBILE WEBSITES ARE FROM THE PAST

I'm not saying that you should not design responsively or that you should go with an `m.*` subdomain. In fact, with social sharing everywhere now, assigning one URL per document, regardless of device, is smart. But this doesn't mean that a single URL should always deliver the same document or that every device should download the same resources.

Let me quote Ethan Marcotte<sup>95</sup>, who coined the term "responsive web design":

*Most importantly, responsive web design isn't intended to serve as a replacement for mobile web sites.*

*— Ethan Marcotte*

---

<sup>93</sup>. <http://www.guypo.com/mobile/rwd-ratio-in-top-100000-websites-refined/>

<sup>94</sup>. <http://www.guypo.com/uncategorized/real-world-rwd-performance-take-2/>

<sup>95</sup>. <http://www.abookapart.com/products/responsive-web-design>

## *Responsive, Mobile And Fast*

We can gain the benefits of responsive design without affecting performance on mobile if we use certain other techniques as well. Responsive web design was never meant to “solve” performance, which is why we can’t blame the technique itself. However, believing that it will solve all of your problems, as many seem to do, would be wrong.

Designing responsively is important because we need to deal with a range of viewport sizes across desktop and mobile. But thinking only of screen size underestimates mobile devices. While the line between desktop and mobile is getting blurrier, different possibilities are still open to us based on the device type. And we can’t decide on functionality using media queries yet.

Some commentators have called this “responsible responsive web design,” while others consider it responsive web design with a modern vision. Without getting into semantics, we do need to understand and be aware of the problem.

While there is no silver bullet and no solutions that can be applied to every type of document, we can use a couple of tricks to improve our existing responsive solutions and maximize performance:

- Deliver each document to all devices with the same URL and the same content, but not necessarily with the same structure.
- When starting from scratch, follow a mobile-first approach.

- Test on real devices what happens when resources are loaded and when `display: none` is applied. Don't rely on resizing your desktop browser.
- Use optimization tools to measure and improve performance.
- Deliver responsive images via JavaScript while we wait for a better solution from browser vendors (such as `src-set`).
- Load only the JavaScript that you need for the current device with conditional loading, and probably after the `onload` event.
- Inline the initial view of a document for mobile devices, or deliver above-the-fold content first.
- Apply a smart responsive solution with one or more of these techniques: conditional loading, responsiveness according to group, and a server-side layer (such as an adaptive approach).

## CONDITIONAL LOADING

Don't always rely on media queries in CSS because browsers will load and parse all of the selectors and styles for all devices (more on this later). This means that a mobile phone would download and parse the CSS for larger screens. And because CSS blocks rendering, you would be wasting precious milliseconds over a cellular connection.

Replace CSS media queries with a JavaScript `matchMedia` query on devices whose context you know will not change. For example, we know that an iPhone

cannot convert to the size of an iPad dynamically, so we would just load the CSS for it that we really need.

We can also use feature detection, such as with Modernizr<sup>96</sup>, to make smarter decisions about the UI and functionality based not only on screen dimension.

## RESPONSIVENESS ACCORDING TO GROUP

While we can rely on a single HTML base and responsive design for all screens when dealing with simple documents, delivering the same HTML to desktops and smartphones is not always the best solution. Why? Again, because of performance on mobile.

Even if we store the same document server-side, we can deliver differences to the client based on the device group. For example, we could deliver a big floating menu to screens 6 inches and bigger and a small hamburger menu to screens smaller than 6 inches. Within each group, we could use responsive techniques to adapt to different scenarios, such as switching between portrait and landscape mode or varying between iPhones (320 pixels wide), 5-inch Android devices (360 pixels) and phablets (400 pixels and up).

## SERVER-SIDE LAYER

The last optional part of a smarter responsive solution is the server. Server-side feature detection and decisions are

---

<sup>96</sup>. <http://modernizr.com/>



not new to the mobile web. Libraries such as WURFL<sup>97</sup> and Device Atlas<sup>98</sup> have been on the market for years.

Mixing responsive design with server-side components is not new. Known sometimes as responsive design and server-side components<sup>99</sup> (RESS) and sometimes as adaptive design, it improves responsive design in speed and usability, while keeping a single code base for everyone server-side.

Unfortunately, these techniques haven't gained much traction in the community over the last few years. Just look at any blog or magazine for developers and compare mentions of "RESS" and "adaptive" to "responsive." There's a reason for that: We are front-end professionals. Anything that involves the server looks like a problem to us, and we don't want that.

In some cases, the front-end designer will be in control of a script on the server; in other cases, a remote development team will manage it, and the designer won't want to deal with the team every time they want to make a small change to the UI. I know the feeling.

That's why it might be time to think of a new architecture layer in large projects, whereby a front-end engineer can make decisions server-side without affecting the back-end architecture. Node.js is an excellent candidate for this platform, being a server-side layer between the current enterprise back-end infrastructure and the front end.

---

<sup>97</sup>. <http://www.scientiamobile.com/>

<sup>98</sup>. <https://deviceatlas.com/>

<sup>99</sup>. <http://www.lukew.com/ff/entry.asp?1392>

In this new layer, the front-end engineer would be in charge of decisions based on the current context that would make the experience fast, responsive and usable on all the devices, without touching the back-end architecture.

## *Responsive Design, Performance And Technical Data*

You might have some doubts by this point. Let's review some technical details to allay your concerns.

Responsive design usually entails delivering the same HTML document to all devices and using media queries to load different CSS and image files. You might have a slightly different idea of what it means, but that is usually how it is implemented.

You might also think that mobile networks today are fast enough to deliver a great experience. After all, 4G is fast, and devices are getting faster.

Well, let's see some data before drawing conclusions.

### **CELLULAR CONNECTIONS**

4G networks are not available everywhere, and even if the whole world was on a 4G network today, the situation might not be what you expect. Less than 3% of mobile phones<sup>100</sup> out there are on a 4G connection. Taking only the US, the number of 4G users has reached approximate-

---

<sup>100</sup>. <http://www.4gamericas.org/index.cfm?fuseaction=page&pageid=2253>

ly 22%, and even those lucky users are not on 4G 40% of the time<sup>101</sup>.

We usually think of mobile network speeds in terms of bandwidth. With 3G, we get up to 5 Mbps; with 4G, we get up to 50 Mbps. But that's not usually the most important factor in a mobile web browsing experience. While more bandwidth is useful for transferring big files (such as a YouTube video), it doesn't add much value when you're downloading a lot of small files and the latency is high and fixed. Latency is the round-trip time that the first byte of every package takes to get to a device after a request.

Cellular networks have more latency than other connections. While the latency on a DSL connection in a US home is between 20 and 45 milliseconds, on 3G it can be between 150 and 450 milliseconds, and on 4G between 100 and 180. In other words, latency is 5 to 10 times higher on a cellular connection than on a home network.

Other issues include the latency when there is a change in radio state, the dead time when a phone turns on the radio to get more data after having been asleep, the lower available memory on average devices and, of course, battery and CPU usage.

## **RESPONSIVE DESIGN ON CELLULAR NETWORKS**

Consider a real case. Keynote, a company that offers performance solutions, has published data on the website

---

<sup>101</sup>. <http://opensignal.com/reports/state-of-lte-q1-2014/>

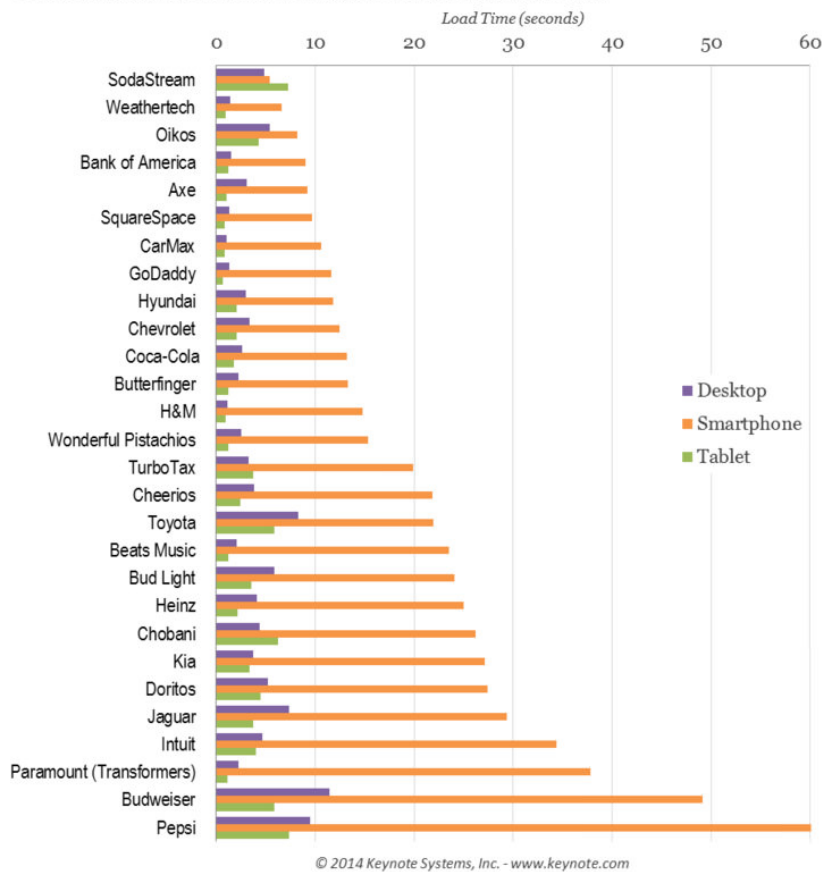
performance of top Super Bowl 2014 advertisers<sup>102</sup>. The data speaks for itself: On a wired or Wi-Fi connection, the loading times range from 1 to 10 seconds, while on a cellular connection, the loading times range from 5 to 60 seconds. Think about that: one full minute to load a website being advertised in the Super Bowl!

### Keynote 3-screen Web Performance Super Bowl 2014 Advertisers



Measurement period: 12:00pm – 8pm PST, 2-Feb 2014

Screens: Desktop (Internet Explorer 9 browser), iPad 2 (high-speed connection), iPhone 5 (wireless carrier)



*Website performance of the top Super Bowl 2014 advertisers.*

<sup>102</sup>. [http://blogs.keynote.com/the\\_watch/2014/02/you-can-run-but-you-cant-hide-from-performance.html](http://blogs.keynote.com/the_watch/2014/02/you-can-run-but-you-cant-hide-from-performance.html)

The same report shows that 43% of those websites offer a mobile-specific version, with an average size of 862 KB; 50% deliver a responsive solution, with an average size of 3211 KB (nearly four times larger); and 7% offer only the desktop version to mobile devices. So, by default, responsive websites are larger than mobile-specific websites.

Of course, responsive design can look different, but, unfortunately, the average responsive website out there looks like these ones of Super Bowl advertisers.

## CLOUD-BASED BROWSERS

If you still doubt that performance is a problem on the mobile web, consider that browser vendors are creating cloud-based browsers to help users — including Opera Mini, the Asia-based UC Browser (which commands 11% of the global market share, according to StatCounter<sup>103</sup>), Amazon Fire’s Silk and now Google Chrome (through a settings option).

These vendors compress every website and resource in the cloud, and then the browser downloads an optimized version to the mobile device. They do it because they know that performance matters a lot to the user’s happiness.

## UNDERESTIMATING THE MOBILE WEB

The web community has always underestimated the importance of mobile browsers. I’m used to hearing people say that the mobile web today is just Safari for iOS and

---

<sup>103</sup>. [http://gs.statcounter.com/#mobile\\_browser-ww-monthly-201303-201403](http://gs.statcounter.com/#mobile_browser-ww-monthly-201303-201403)

Chrome for Android and that, for responsive design, we need only care about viewports that are 320 pixels wide. It's far more complex than that.

Today, more than 10 browsers have a market share over 1%. Even if you want to consider only the default browsers on iOS and Android, it's not so simple. Roughly speaking<sup>104</sup>, 50% of mobile users browse the web on iOS, 38% on Android, 3% on Windows Phone, 5% with Opera Mini (on various operating systems) and 4% on other platforms.

On Android, 64% of users today browse with Android's stock browser, which is not the same as Google Chrome and which exists in different versions. Moreover, some of Samsung's latest Galaxy devices have a version of the Android browser with a customized engine.

In terms of viewport size, we are dealing today with pixel widths of 320, 360, 400 and 540 with Android smartphones alone. My suggestion, then, is never to underestimate the mobile web, and to learn its unique characteristics.

## **ABOVE-THE-FOLD CONTENT IN 1 SECOND**

On a mobile device, we can consider a website to be fast when content above the fold (i.e. the content that is visible without scrolling) is rendered in 1 second or less. I know, 1 second seems awfully fast — especially considering that at least half of that time is taken up by the cellular connection — but it has been proven to be possible. A

---

<sup>104</sup>. <http://firt.mobi/velocity>

1-second response keeps users engaged with the content, thereby increasing the conversion rate and reducing abandonments.

To achieve a 1-second response time, above-the-fold content needs to be received in one round trip over the transmission control protocol (TCP) — remember that the average 3G connection has almost half a second of latency. Because of a TCP feature known as a “slow start,” that first response should be no more than about 14 KB in order to avoid a second package. This means that at least the HTML and CSS for the above-the-fold content should fit in a single 14 KB HTTP response. If we achieve that, then we’ll have achieved the perception of a 1-second loading time.

This rule is not written in stone and will vary based on your content. However, because content that appears above the fold will usually not be the same on a mobile screen as on a desktop screen, achieving this goal of 1 second with a responsive design is very difficult. It’s possible, but combining techniques makes it much easier.

## ONE HTML FOR ALL

A typical responsive design delivers a single HTML document to all devices: televisions, desktops, tablets, smartphones and feature phones. It sounds great, but we live in a world that has cellular and other problems. Your responsive HTML might render correctly on mobile devices, but it’s not as fast as it should be, and that is affecting your conversion rate.

If a single `display: none` appears in any of your CSS, then your website is not as fast as it could be. On a website that has been designed from scratch to be semantic, then the responsive overload would be almost nil; on a website whose HTML includes 40 external scripts, jQuery plugins and fancy libraries, mostly for the benefit of big screens, then the responsive overload would be at the high end. When the same HTML is used, then the same external resources would be declared for all devices.

This isn't to say that responsive design alone can't be done, just that the website won't be optimized by default. If you are sensitive to performance, then your responsive solution might look different than the usual.

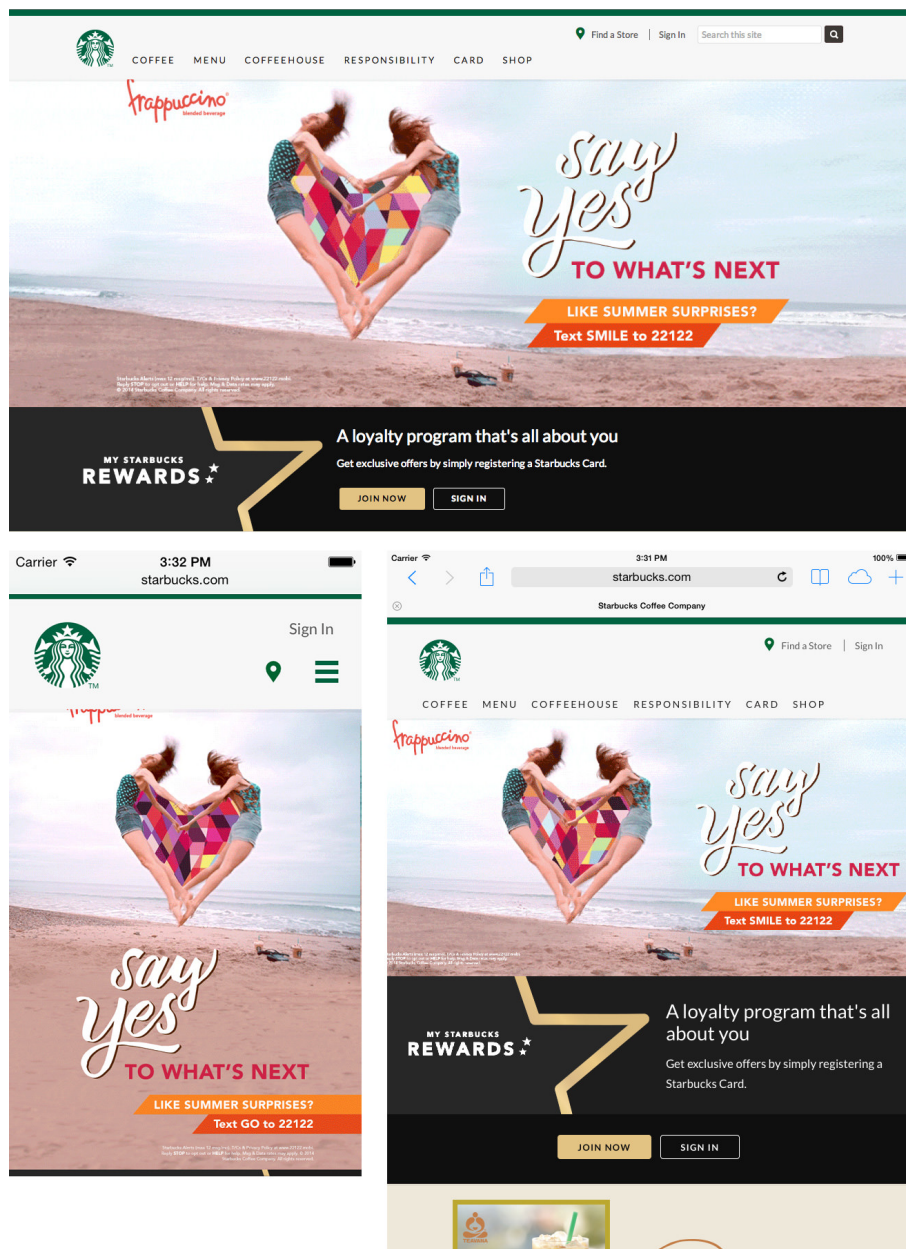
Let's review Starbucks' website. Its home page is responsive and looks great in the three viewports we tested (see the screenshots below). But upon checking the internals, we see that all versions load the same 33 external JavaScript files and 6 CSS files. Does a mobile device with 3G latency deserve 39 external files just to get the view shown below?

You might be thinking, "Hey, blame the implementation, not the technique<sup>105</sup>." You're right. This article is not against responsive web design. It's against aiming for responsiveness in a way that leads to a weak implementation, and it's against prioritizing responsiveness over performance, as we see with Starbucks. It looks great when you resize the browser, but that's not all that is important. Performance also matters a lot to mobile users.

---

<sup>105</sup> <http://timkadlec.com/2012/10/blame-the-implementation-not-the-technique/>





The Starbucks's website in different states.

If your responsive website has performance problems, then the fault may lie with how you've framed the goal. If you have the budget for responsive design, then you must also have the budget for performance.

## LOADING RESOURCES

Media queries are implemented in different ways, usually as one of the following:

- a single CSS file with multiple `@media` declarations,
- multiple CSS files linked to from the main page via media attributes.

In the first case, every device would load the CSS intended for all devices because there would be just one CSS group. Hundreds of selectors that will never be used are transferred and parsed by the browser anyway.

You might think that multiple external files are better because the browser would load the resources based on breakpoints. This is what we're taught in tutorials in blogs, magazines, books and training courses.

```
<link rel="stylesheet" href="desktop.css"
      media="(min-width: 801px)">
<link rel="stylesheet" href="tablet.css"
      media="(min-width: 401px) and (max-width: 800px)">
<link rel="stylesheet" href="mobile.css"
      media="(max-width: 400px)">
```

Well, you'd be wrong. All browsers will load all external CSS, regardless of context. The screenshot below shows an iPhone downloading all of the CSS files excerpted above, even ones not intended for it.



*Browser will load all external CSS files, regardless of the context.*

Why do browsers download all CSS files? Suppose you have one CSS file for portrait orientation and another for landscape. We wouldn't want browsers to load CSS on the fly when the orientation changes, in case a couple of milliseconds go by without any CSS being used. We'd want the browser to preload both files. That's what happens when you define media queries based on screen dimensions.

Can the dimensions of mobile browsers be changed? Mostly not yet, but vendors are preparing their mobile browsers to be resized like desktop browsers, which is why the browsers usually load all CSS declarations regardless of whether their width matches the media query.

While stretchable viewports don't exist on mobile devices (yet), some viewports resize in certain situations:

- when the orientation changes in certain browsers,
- when the viewport declaration changes dynamically,
- when offset content is added after **onload**,

- when external mirroring is supported,
- when more than one app is open at the same time on some Samsung Android devices (in multi-window mode).

Browsers that are optimized for these changes in context will preload all resources that they might need.

While browsers might be smarter about this in the near future, we're left with a problem now: We are delivering more resources than are needed and, thus, penalizing mobile users for no reason.

## *The Real Problem: Responsive Design As A Goal*

As Lyza Danger Gardner says in “[What We Mean When We Say ‘Responsive’](#)<sup>106</sup>,” designers define “responsive” differently, which can lead to communication problems.

Let's get to the root. The term first appeared in a 2010 post by [Ethan Marcotte](#)<sup>107</sup>, followed by a book with the same name. Ethan defines it as providing an optimal viewing experience across a wide range of devices using three techniques: fluid grids, flexible images and media queries.

Nothing is wrong with that definition. The problem is when we set it as the goal of a website without understanding the broader goals that we need to achieve.

---

<sup>106</sup>. <http://alistapart.com/column/what-we-mean-when-we-say-responsive>

<sup>107</sup>. <http://alistapart.com/article/responsive-web-design/>

When you set responsive design as a goal, it becomes easy to lose perspective. What is the real problem you are trying to solve? Is being responsive really a problem? Probably not. But do you understand “being responsive” to mean “being mobile-compatible”? If so, then you might be making some mistakes.

The ultimate goal for a website should be “happy users,” which will lead to more conversions, whatever a conversion might be, whether it’s getting a visitor to spread the word, providing information or making a sale. Users won’t be happy without a high-performing website.

The direct impact of performance on conversions, particular in mobile, has been proven many times. If this is the first time you are hearing about this, just check any of Steve Souders<sup>108</sup>’ expert books about optimizing web performance.

When you know your goals, you can decide which tools and techniques are best to achieve them. This is when you analyze where and how to use a responsive approach. You *use* responsive design — you don’t *achieve* it.

## RESPONSIVE VS. USERS

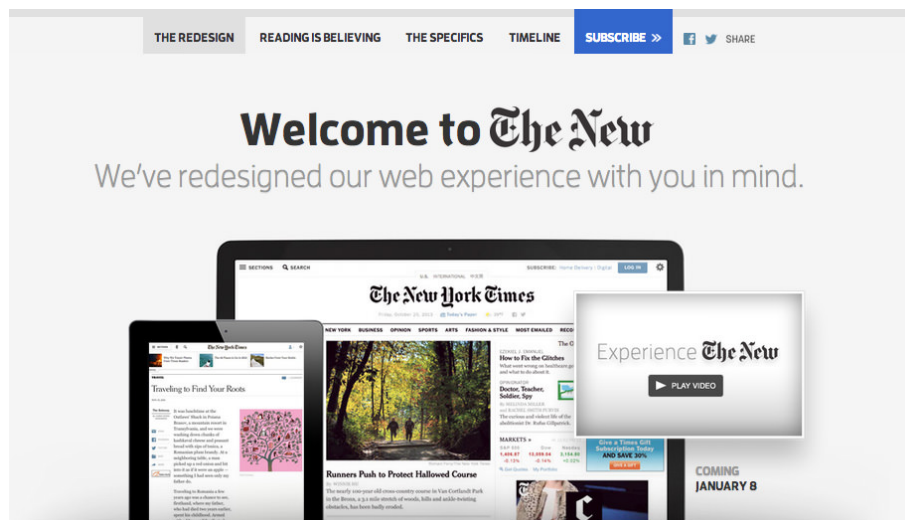
The New York Times redesigned its website<sup>109</sup> a couple of months ago with the goal of keeping “you in mind.” Meanwhile, thousands of other big companies present their new responsive websites with pride.

---

<sup>108</sup>. <http://stevesouders.com>

<sup>109</sup>. <http://www.nytimes.com/redesign/>

The New York Times follows responsive design in different ways, but some people complained that it still uses a separate mobile version, instead of adapting the layout based on the same HTML. An article even came out titled “The Latest New York Times Web App Misses the Point of Responsive Design<sup>110</sup>.”



**Sleeker. Faster.  
More intuitive.**  
In a word, "enhanced."

A more immersive reading experience? We're glad you asked. We've streamlined our article pages and created a more responsive interface with faster load times. So navigating between stories is easier and finding more content that appeals to you is just a click, swipe or tap away.

*The New York Times follows responsive design in different ways.*

Who said that responsive web design means supporting all possible screen sizes with the same HTML? Sure, this is a common understanding, but that rule isn't written anywhere. It's just our need to simplify the problem that has led to it.

---

<sup>110</sup>. <http://readwrite.com/2013/12/05/new-york-times-responsive-web-app-todays-paper>

In recent months, companies have said things along the lines of, “We’ve applied a new responsive design, and now our mobile conversions have increased by 100%.” But did conversions increase because the website was made to be responsive, or are users realizing that the website is now responsive and so are happier and convert more?

People convert more because their experience on mobile devices is now better and faster than whatever solution was in place before (whether it was a crude mobile version or a crammed-in desktop layout). So, yes, responsiveness is better than nothing and better than an old mobile implementation. But a separate mobile website with the same design or even a smarter solution done with other techniques would achieve the same conversion rate or better.

## Conclusion

*Your visitors don’t give a sh\*t if your site is responsive.*

— Brad Frost

Brad Frost<sup>111</sup> is completely right. Users want something fast and easy to use. They don’t usually resize the browser, and they don’t even understand what “responsive” means.

It’s a bitter truth, and it doesn’t quite apply to all websites. But it’s better than thinking, “We can relax. Our website is responsive. We’ve taken care of mobile.” Some-

---

<sup>111</sup>. <http://bradfrostweb.com/blog/web/responsive-web-design-missing-the-point/>



times, even when not relevant to the situation, saying that responsive design is “bad for performance<sup>112</sup>” can be good because it helps to spread the word on why performance is so important.

The New York Times is right: The goal is the user. It’s not a tool or a technique or even the designer’s happiness.

## FURTHER RESOURCES

- “Responsive Web Design: Missing the Point<sup>113</sup>,” Brad Frost
- “RESS: Responsive Design + Server-Side Components<sup>114</sup>,” Luke Wroblewski
- “What We Mean When We Say ‘Responsive’<sup>115</sup>,” Lyza Danger Gardner, A List Apart
- “You Can Run, But You Can’t Hide From Performance<sup>116</sup>,” Aaron Rudger, Keynote
- “Real-World RWD Performance: Take 2<sup>117</sup>,” Guy Podjarny
- “Blame the Implementation, Not the Technique<sup>118</sup>,” Tim Kadlec

---

<sup>112</sup>. [http://timkadlec.com/2014/07/](http://timkadlec.com/2014/07/rwd-is-bad-for-performance-is-good-for-performance/)

[rwd-is-bad-for-performance-is-good-for-performance/](http://timkadlec.com/2014/07/rwd-is-bad-for-performance-is-good-for-performance/)

<sup>113</sup>. <http://bradfrostweb.com/blog/web/responsive-web-design-missing-the-point/>

<sup>114</sup>. <http://www.lukew.com/ff/entry.asp?1392>

<sup>115</sup>. <http://alistapart.com/column/what-we-mean-when-we-say-responsive>

<sup>116</sup>. [http://blogs.keynote.com/the\\_watch/2014/02/](http://blogs.keynote.com/the_watch/2014/02/)

[you-can-run-but-you-cant-hide-from-performance.html](http://blogs.keynote.com/the_watch/2014/02/you-can-run-but-you-cant-hide-from-performance.html)

<sup>117</sup>. <http://www.guypo.com/uncategorized/real-world-rwd-performance-take-2/>

<sup>118</sup>. <http://timkadlec.com/2012/10/blame-the-implementation-not-the-technique/>



- “RWD Is Bad for Performance’ Is Good for Performance<sup>119</sup>,” Tim Kadlec 🐘

---

<sup>119</sup>. <http://timkadlec.com/2014/07/rwd-is-bad-for-performance-is-good-for-performance/>

# How To Make Your Websites Faster On Mobile Devices

BY JOHAN JOHANSSON 🍷

A recent study<sup>120</sup> (PDF) found that more than 80% of people are disappointed with the experience of browsing Web on mobile devices and would use their smartphones more if the browsing experience improved.

This isn't surprising when 64% of smartphone users expect websites to load in 4 seconds or less, while the average website takes more than twice that amount, at 9 seconds. This chapter explains techniques you can use to make your websites faster on mobile devices.



*User experience on mobile devices usually has room for improvement.*

*(Image: Phil Campbell<sup>121</sup>)*

---

<sup>120</sup>. <http://www.keynote.com/docs/reports/Keynote-2012-Mobile-User-Survey.pdf>

<sup>121</sup>. <http://www.flickr.com/photos/clanlife/6369792721/>

## Download Speeds For Mobile Users

Let's start by looking at what influences the loading speed of a website on a smartphone.

The most obvious factor is the connection speeds of smartphones. In the best-case scenario, mobile users connect to the Internet over 3G and 4G networks, with 4G networks being faster. In the US, 57% of users are on 3G, and 27% are on 4G. In Canada, 4G penetration is even lower. In the UK, 4G only recently became available.

According to a study by PCWorld<sup>122</sup>, the average download speed for 3G networks in the US is 2 Mbps, and 6.2 Mbps for 4G networks. A study by Ofcom found that the average download speed for 3G in the UK to be 2.1 Mbps. Outside of North America and Europe, connection speeds are generally slower. Because 1 Mbps equals 128 KB/s (or 0.128 MB/s), this translates into the following:

- 256 KB/s on average for 3G users (0.256 MB/s),
- 793 KB/s on average for 4G users (0.793 MB/s).

If you multiply that by the 4 seconds that mobile users are expecting to wait, this means the website could be a maximum of 1 MB for 3G users and 3 MB for 4G users.

However, download speed is not the bottleneck. The bottleneck is the network latency<sup>123</sup>, smartphone's memory and CPU. Even if the phone can download 1 MB in 4

---

<sup>122</sup>. <http://www.pcworld.com/article/254888/>

[3g\\_4g\\_performance\\_map\\_data\\_speeds\\_for\\_atandt\\_sprint\\_t\\_mobile\\_and\\_verizon.html](http://www.pcworld.com/article/254888/3g_4g_performance_map_data_speeds_for_atandt_sprint_t_mobile_and_verizon.html)

<sup>123</sup>. <http://www.igvita.com/2012/07/19/>

[latency-the-new-web-performance-bottleneck/](http://www.igvita.com/2012/07/19/latency-the-new-web-performance-bottleneck/)

seconds, the website will take longer to load because the phone needs to receive and process the code and images.

On a desktop, only 20% of the time it takes to display a Web page comes from downloading files. The rest of the time is spent processing HTTP requests and loading style sheets, script files and images. It takes even longer on a smartphone because its CPU, memory and cache size are much smaller than a desktop's.

## *How To Minimize Loading Time*

Having a fast website is all about making the hard decisions and getting rid of what's not at the core of your experience. If it doesn't add a lot of value, remove it. This is true for all phases of the development process, but especially so for planning and coding.

- **Reduce Dependencies**

Fewer files to download means fewer HTTP requests and faster loading times.

- **Reduce Image Dimensions**

On top of the extra download time, precious processing power and memory are used to resize high-resolution images.

- **Reduce Client-Side Processing**

Rethinking the use of JavaScript and keeping it to a minimum are best.

# How To Reduce Dependencies

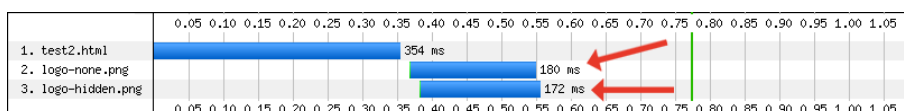
## LOAD IMAGES THROUGH CSS

If you want to hide content images from mobile users, relying on **display: none** or **visibility: hidden** won't prevent them from being downloaded. We tested the following code:

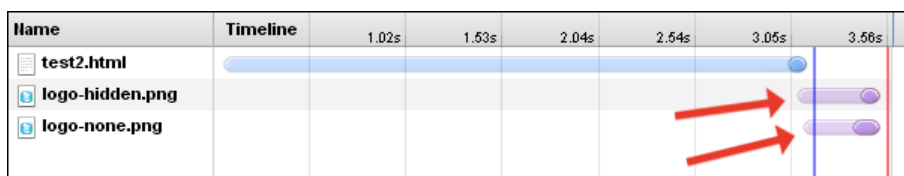
```
<div style="display:none;">
  
</div>

<div style="visibility:hidden;">
  
</div>
```

You can see in the two waterfall charts below how content images set to **display: none** or **visibility: hidden** are still downloaded.



Waterfall chart for **display: none** and **visibility: hidden** on an iPhone 4, iOS 5.0.



Waterfall chart for **display: none** and **visibility: hidden** on a Nexus S.

Instead, load them as background images in CSS, and use media queries to conditionally hide them. The basis for this technique was originally tested by Jason Grigsby<sup>124</sup> and expanded upon by Tim Kadlec. A variant is used on Amazon's separate mobile pages<sup>125</sup> to conditionally load device-specific images.

```
<meta name="viewport" content="width=device-width">
```

```
<style>
@media (max-width:600px) {
    .image {
        display:none;
    }
}
@media (min-width:601px) {
    .image {
        background-image: url(image1.jpg);
    }
}
</style>
```

```
<div class="image"></div>
```

You can see in the two waterfall charts below that the image isn't loaded:

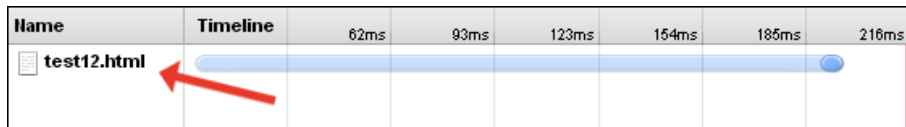
---

<sup>124</sup>. <http://blog.cloudfour.com/css-media-query-for-mobile-is-fools-gold/>

<sup>125</sup>. <http://www.amazon.com/gp/aw/d/0345535421>



Waterfall chart for an iPhone 4, iOS 5.0.



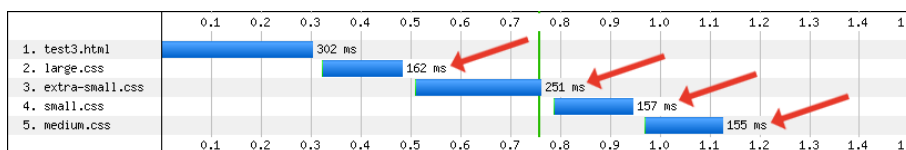
Waterfall chart for a Nexus S.

## KEEP EXTERNAL STYLE SHEETS TO A MINIMUM

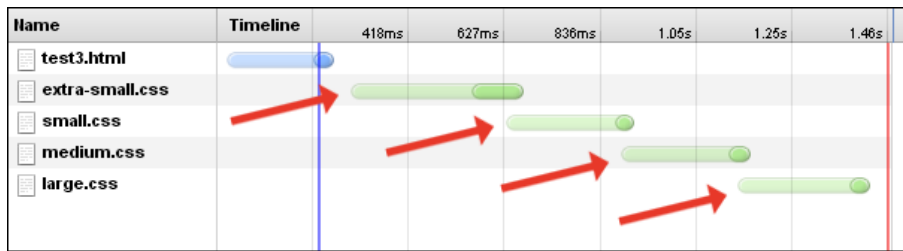
If you've been using separate style sheets for each break point, you may want to rethink this. We tested the following code:

```
<link href="extra-small.css" rel="stylesheet" media="screen
and (max-width: 390px)" />
<link href="small.css" rel="stylesheet" media="screen and
(min-width: 391px) and (max-width: 500px)" />
<link href="medium.css" rel="stylesheet" media="screen and
(min-width: 501px) and (max-width: 767px)" />
<link href="large.css" rel="stylesheet" media="screen and
(min-width: 768px)" />
```

You can see that all four style sheets are downloaded in portrait mode:



Waterfall chart for media queries on an iPhone 4, iOS 5.0.



*Waterfall chart for media queries on a Nexus S.*

Because they're all being downloaded anyway, you might as well combine them all into a single file and reduce the number of HTTP requests. Alternatively, you could rely on server-side functions to dynamically insert the correct style sheet based on the device (a method used on the responsive [WordPress.com](http://wordpress.com)<sup>126</sup>).

A different approach would be to use inline styles. Amazon's separate mobile product pages have one external 6 KB style sheet, along with some inline styles. This results in a single additional HTTP request to download all page styles. Amazon's desktop version isn't as efficient, with nine external style sheets, totalling 40 KB combined.

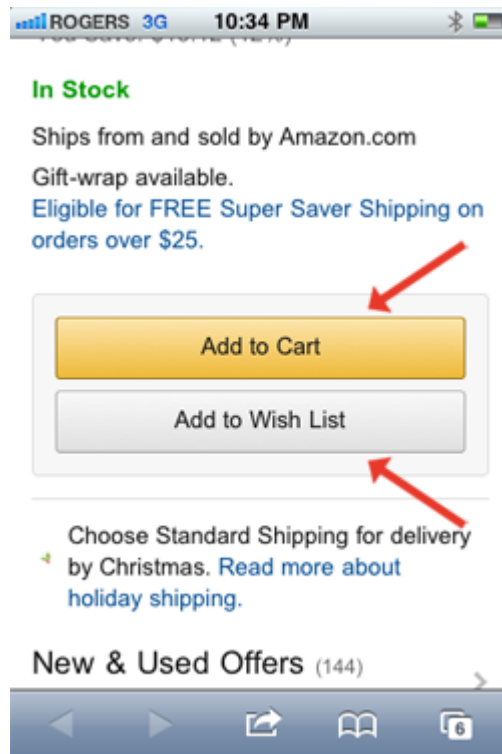
## CSS<sub>3</sub> INSTEAD OF IMAGES

Rounded corners, drop shadows, gradient fills, and so on – these styling features can be used instead of images, reducing the number of HTTP requests and speeding up loading time.

---

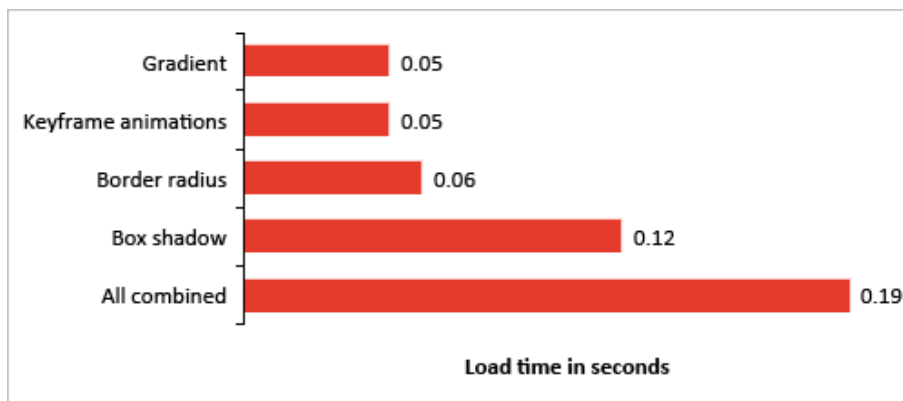
<sup>126</sup>. <http://wordpress.com>





*Amazon's buttons are created using CSS3; no images are used.*

Although CSS3 can reduce HTTP requests, it adds to the processing load. We created a series of simple HTML files, each containing one basic CSS3 styling feature. You can see from the chart below that the effect on loading time is minimal but should still be considered. Notice how the box-shadow effect has the biggest impact on loading time.



*Loading times for CSS3 styling features on an iPhone 4, iOS 5.0.*

## DATA URI INSTEAD OF IMAGES OR WEB FONT FILES

The data URI scheme<sup>127</sup> is a way to embed data into HTML or CSS without using any external resources. It can be used to embed anything onto a Web page, with the most common usage being to embed images and Web font files. Its primary benefit is to reduce the number of HTTP requests.

The way it works is simple. Instead of referencing an external image file, you would embed the base64-encoded data directly into the HTML or CSS, using the following format:

```
data:[MIME-type][;charset=encoding][;base64],[data]
```

For example, the following shortcut icon is generated by a data URI:

---

<sup>127</sup>. [http://en.wikipedia.org/wiki/Data\\_URI\\_scheme](http://en.wikipedia.org/wiki/Data_URI_scheme)



Here's the code:

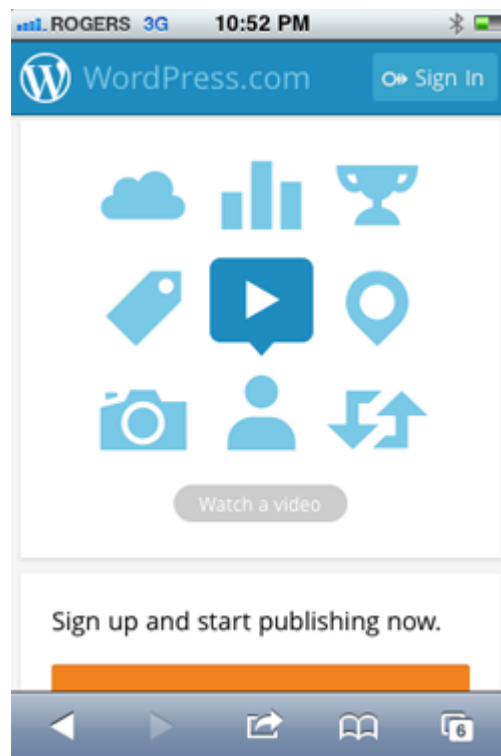
```

```

The WordPress.com responsive website embeds both images and fonts. The Boston Globe<sup>128</sup>'s responsive website, which loads in just over 4 seconds on a smartphone, also embeds data.

---

<sup>128</sup>. <http://www.bostonglobe.com/>



*Nearly all graphics and Web fonts on WordPress.com are from data URIs.*

Because of the way browsers load data URIs, they could end up taking longer to load than using external image or font files. Testing and comparing the two methods is important to see whether it's worthwhile.

## **INLINE SCALABLE VECTOR GRAPHICS (SVG) INSTEAD OF IMAGES**

Much like data URIs, scalable vector graphics<sup>129</sup> (SVG) can be embedded onto a page to reduce the number of HTTP requests. For example, the following image is an inline SVG:

---

<sup>129</sup>. [http://en.wikipedia.org/wiki/Scalable\\_Vector\\_Graphics](http://en.wikipedia.org/wiki/Scalable_Vector_Graphics)



Here's the code:

```
<svg version="1.1" id="drop" x="0px" y="0px"
  width="17.812px" height="28.664px"
  viewBox="296.641 381.688 17.812 28.664"
  enable-background="new 296.641 381.688 17.812
  28.664"
  xml:space="preserve">
<path fill="#EE1C4E" d="M314.428,401.082c-0.443-5.489-5.146-
9.522-7.52-14.186c-0.816-1.597-1.352-5.208-1.352-5.208
s-0.555,3.792-1.388,5.425c-2.233,4.371-7.127,8.999-7.507,
14.047c-0.36,4.794,4.101,9.191,8.896,9.191
C310.49,410.354,314.816,405.941,314.428,401.082z"/>
</svg>
```

SVG files can be created with a vector graphics editor, such as Adobe Illustrator. Once created, open the file in a text editor and drop it into the code (minus any unnecessary meta data).

The above code will work in an HTML file, but won't work in a style sheet. To embed an SVG file in a style sheet, first convert it to a data URI. To do this, grab the SVG code from the text editor (be sure to include the meta data), encode it in base64, and then embed it using the following format:

```
data:image/svg+xml[;base64],[data]
```

Here's the code:

```
background-image:url(data:image/svg+xml;base64,PD94bWwgdmVyc2
1vbjo0MS4wIiBlbmNvZGluc20idXRmLTgiPz4NCjwhLS0gR2VuZXJhdG9yOjBB
ZG9iZSBjbG1c3RyYXRvciAxNS4xLjAsIFNWRyBFbVcnQgUGx1Zy1JbiAuIF
NWRyBWZXJzaW9uOAA2LjAwIEJ1aWxkIDApICAuLT4NCjwhRE9DVFlQRSBzdmcg
UFVCTELDIICiLy9XM0Mv0RURCBTVkc9MS4xLy9FTiIgImh0dHA6Ly93d3cudz
Mub3JnL0dyYoaWNzL1NWRy8LjEvRFREL3N2ZzExLmR0ZCQo8c3ZnIHZlcnN
pb249IjEuMSIgaWQ9IkxheWVyXzIiIHhtbG5zPSJodHRwOi8vd3d3LnczLm9yZ
y8yMDAwL3N2ZyIgeG1sbnM6eGxpbnM9Ih0dHA6Ly93d3cudzMub3JnLzE5OTk
veGxpbnMsiIHg9IjBweCIgeT0iMHB4IgoKCSB3WR0aD0iMTcuODEycHgiIGhla
WdodD0iMjguNjY0cHgiIHZpZXh0c3g9IjI5Ni42NDEMzgXlY4OCAxNy44MTI
gMjguNjY0IgoKCSBlbmFibGUtYmFja2dyb3VuZD0ibmV3ID5Ni42NDEgMzgXl
jY4OCAxNy44MTIgaWQ9IjB4bWw6c3BhY2U9InByZXNlcnZlI4NCjxwYXR
oIGZpbGw9IiNFRTFDNEUiIGQ9Ik0zMtQuNDI4LDQwMS4wODJjLTAuNDQzTUuN
Dg5LTUuMTQ2LTkuNTIyLTcuNTI0tMTQuMTg2Yy0wLjgXNi0xLjU5Ny0xLjM1Mi
0LjIwOC0xLjM1Mi01LjIwOA0KCXMtMC41NTUsMy43OTI0tMS4zODGsNS40MjVj
LTIuMjzLDQwMzcXLTcuMTI3LDguOTk5LTcuNTA3LDE0LjA0N2MtMC4zNi0wLj
c5NCw0LjEwMw5LjE5MSw4Ljg5Ni0wLjE5MQ0KCUMzMTAuNDksNDEwLjM1NCwz
MTQuODE2LDQwNS45NDEsMzE0LjQyOCw0MDgyeiIvPg0KPC9zdmc+DQo=);
```

Test this method and compare with external image files to make sure it actually is faster.

## IMAGE SPRITES

The idea behind sprites is to combine commonly used images into a single image file, reducing the number of HTTP requests. For example, if you combine four images into a single sprite, you're theoretically reducing HTTP requests from four to one. The required image is then displayed by using the CSS **background-position** property.



*One of Amazon's image sprites.*

Amazon has multiple sprites, some with duplicated images.

## FONT ICONS

Font icons are fonts consisting of symbols and glyphs (like Wingdings or Webdings), and can be used instead of loading an image file. For example, the following icon is not an image, but rather the letter “H” from Wingdings:

H

Although Wingdings and Webdings are a bit cheesy, other more professional Web fonts are available that can be loaded through the `@font-face` rule.

This technique can be used in the same way as image sprites to reduce HTTP requests. By combining multiple icons into a single Web font, the number of HTTP requests for all icons can be reduced to one. If the Web font is embedded using a data URI (as described above), HTTP requests could be reduced to zero.

This is exactly what WordPress.com does. Here's the Web font embedded in its style sheet:



*WordPress.com's font icons*

WordPress.com has access to all of these icons without having to make any extra HTTP requests, because the icons are part of a Web font embedded in WordPress' style sheet as a data URI.

As a bonus, font icons can be animated using CSS3 keyframe animation (which would be useful for "loading" icons).

The primary downside of CSS sprites is that they can only be one solid color. Amazon's image sprites include multi-colored icons, which is why it couldn't use this technique.



Tools such as [IcoMoon](http://icomoon.io)<sup>130</sup> make it easy to build a custom Web font. All that's needed is the SVG file for each of the icons.

## AVOID INLINE FRAMES

Each inline frame (iframe) results in one more HTTP request, in addition to any dependencies within the iframe. Here's a quick test we did, comparing inline text with a single iframe containing text only.



*Loading times for CSS3 styling features on an iPhone 4, iOS 5.0.*

Having a single iframe added nearly 0.20 seconds to the loading time. To keep the website fast, it's best not to use them.

## CODE FOR MOBILE-FIRST

Going mobile-first also applies to front-end development.

By coding for mobile users first and then progressively enhancing for tablets and desktops, unnecessary dependencies are reduced. Compare this to coding for desktop first, where heavy components are loaded by default

---

<sup>130</sup>. <http://icomoon.io>

and then hidden for small screens (known as “graceful degradation”).

Here’s an example of coding for desktop first:

```
<style>
.image {
    background-image: url(image1.jpg);
}

@media (max-width:390px) {
    .image {
        display: none;
    }
}
</style>

<div class="image"></div>
```

In the above code, the default is to display the image, which is then overruled for mobile devices with the media query.

Here’s an example of coding for mobile first:

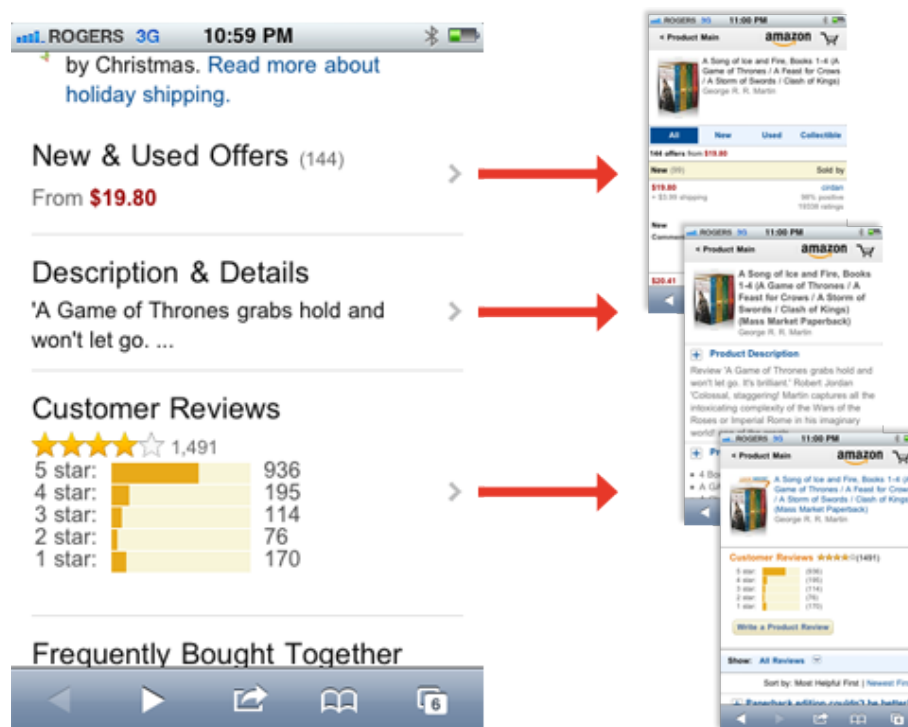
```
<style>
@media (min-width:391px) {
    .image {
        background-image: url(image1.jpg);
    }
}
</style>
```

<div class="image"></div>

By default, the image isn't displayed, while wider screens are progressively enhanced using a media query.

## SPLIT CONTENT ONTO MULTIPLE PAGES (SEPARATE MOBILE WEBSITES)

Keep your core content on the page, while linking to secondary content. This will reduce the payload of the HTML and prevent any associated dependencies from being downloaded.



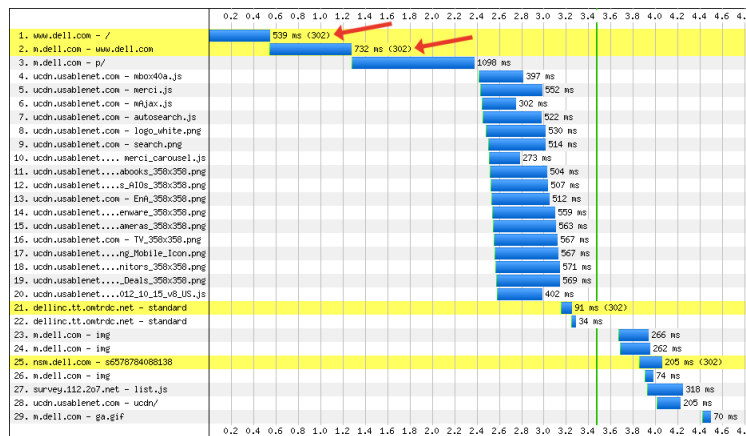
*Amazon splits content onto multiple pages to reduce loading time.*

Amazon's mobile product pages have generic product information, while providing links to "Customer Reviews,"

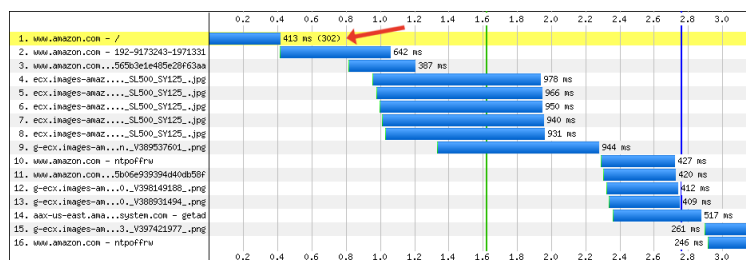
“Description & Details” and “New & Used Offers.” This eliminates HTTP requests for three images, while the HTML’s size is reduced by 45 KB.

## KEEP REDIRECTS TO A MINIMUM (SEPARATE MOBILE WEBSITES)

Amazon has a single redirect taking the visitor to the separate mobile page, resulting in a 0.4 second delay. Compare that with Dell’s website, which has two redirects, resulting in a 1.2 second delay.



Waterfall chart for Dell’s website on an iPhone 4, iOS 5.0.



Waterfall chart for Amazon’s website on an iPhone 4, iOS 5.0.

# How To Reduce Image Dimensions

## RESPONSIVE IMAGES

The idea behind responsive images is to have the visitor download only those images that are best suited to their device. In the case of smartphones, this would be lower-resolution images that can be quickly downloaded and rendered.

Amazon's separate mobile product pages use a responsive-images technique that assigns a particular background image to a **div** according to media-query matches. Here's Amazon's code:

```
<!-- // This meta viewport is inserted for iPhones // -->
<meta name="viewport" content="width=device-width,
user-scalable=no,initial-scale=1.0,maximum-scale=1.0">

<!-- // This meta viewport is inserted for the Nexus S // -->
<meta name="viewport" content="width=device-width">

<style>
@media (max-width:390px) {
    #image-container {
        max-width: 109px;
    }
    .image {
        background-image: url(image1.jpg);
    }
}

@media (max-width:390px) and
```

```
(-webkit-min-device-pixel-ratio:1.5) {  
    .image {  
        background-image: url(image2.jpg);  
    }  
}  
  
@media (max-width:390px) and  
(-webkit-min-device-pixel-ratio:2) {  
    .image {  
        background-image: url(image3.jpg);  
    }  
}  
  
@media (min-width:391px) and (max-width:500px) {  
    #image-container {  
        max-width: 121px;  
    }  
    .image {  
        background-image: url(image4.jpg);  
    }  
}  
  
@media (min-width:391px) and (max-width:500px) and  
(-webkit-min-device-pixel-ratio:1.5) {  
    .image {  
        background-image: url(image5.jpg);  
    }  
}  
  
@media (min-width:391px) and (max-width:500px) and  
(-webkit-min-device-pixel-ratio:2) {  
    .image {  
        background-image: url(image6.jpg);  
    }  
}
```

```

}

@media (min-width: 501px) and (max-width: 767px) {

    #image-container {

        max-width: 182px;

    }

    .image {

        background-image: url(image5.jpg);

    }

}

@media (min-width: 501px) and (max-width: 767px) and
(-webkit-min-device-pixel-ratio:1.5) {

    .image {

        background-image: url(image7.jpg);

    }

}

@media (min-width: 501px) and (max-width: 767px) and
(-webkit-min-device-pixel-ratio:2) {

    .image {

        background-image: url(image8.jpg);

    }

}

@media (min-width:768px) {

    #image-container {

        max-width: 303px;

    }

    .image {

        background-image: url(image8.jpg);

    }

}

@media (min-width:768px) and

```

```

(-webkit-min-device-pixel-ratio:1.5) {
    .image {
        background-image: url(image8.jpg);
    }
}

@media (min-width:768px) and
(-webkit-min-device-pixel-ratio:2) {
    .image {
        background-image: url(image8.jpg);
    }
}

</style>

<div id="image-container">
    <div class="image">
        
    </div>
</div>

```

Even though Amazon has a total of eight product images in its inline styles, only two are downloaded by an iPhone 4 or Nexus S in portrait mode.

A different [data-fullsrc](#) responsive-images technique<sup>131</sup> is used on the Boston Globe's responsive website. It's a combination of markup, JavaScript and a server-side redirect rule:

---

<sup>131</sup>. <http://www.alistapart.com/articles/responsive-images-how-they-almost-worked-and-what-we-need/>



```

```

The `src` is the mobile-sized image, ensuring that the website defaults to the smaller version (mobile-first), while the `data-fullsrc` is the full-sized image. JavaScript is used to detect the device's screen size, which is written to a cookie. For large screen sizes, JavaScript swaps the smaller `src` image with the higher-resolution `data-fullsrc` image. The server also uses Apache rewrite rules to check for `.r.` in the image file's name and displays a spacer GIF, instead of the smaller mobile image (thus preventing the mobile-sized image from being downloaded by desktops).

Microsoft<sup>132</sup>'s responsive website uses Scott Jehl's Picturefill<sup>133</sup> technique:

```
<div data-picture data-alt="Alternate text here">
  <div data-src="image1.png"></div>
  <div data-src="image2.png"
    data-media="(min-device-pixel-ratio: 2.0)">
  </div>
  <div data-src="image3.png" data-media="(max-width:
    539px)"></div>
  <div data-src="image4.png" data-media="(max-width: 539px)
    and (min-device-pixel-ratio: 2.0)"></div>

  <noscript>132</sup>. <http://www.microsoft.com/>

<sup>133</sup>. <https://github.com/scottjehl/picturefill>

```
text here" /></noscript>
</div>
```

With this technique, JavaScript sweeps through the page's code and finds the `div`s with the `data-picture` attribute. It then inserts a new `img` tag based on the `data-media` attribute.

The main benefits of these responsive-image techniques are:

- Small screens download low-resolution images, while large screens download high-resolution images;
- Only the required images are downloaded, while unneeded images aren't loaded in the background.

There are a variety of other techniques for implementing responsive images. Check out these resources for more details:

- “[Responsive IMGs<sup>134</sup>](#),” Jason Grigsby, Cloud Four Blog  
A three-part series on responsive images.
- “[Which Responsive Images Solution Should You Use?<sup>135</sup>](#),” Chris Coyier, CSS-Tricks

---

<sup>134</sup>. <http://blog.cloudfour.com/responsive-imgs/>

<sup>135</sup>. <http://css-tricks.com/which-responsive-images-solution-should-you-use/>

## How To Reduce Client-Side Processing

### KEEP JAVASCRIPT TO A MINIMUM

With JavaScript disabled in Chrome, Starbucks<sup>136</sup>' responsive website takes 3.53 seconds to load on a good broadband connection on desktop. With JavaScript enabled, it takes 4.73 seconds, a 34% increase. JavaScript's impact on loading time would be even greater on a smartphone because of the device's smaller CPU, memory and cache size. As a general rule, rethink whether to use JavaScript, and keep it to a minimum.

A good example of spartan JavaScript is the BBC<sup>137</sup>'s mobile website. The website doesn't use external JavaScript files — it's all inline. The inline script is limited to a few lines and doesn't have a significant impact on memory, with the HTML file and all inline JavaScript taking 0.78 seconds to load.

Much like the BBC, Amazon's mobile product pages don't have external JavaScript files, instead using minimal inline scripts. The HTML file and all inline JavaScript take 0.75 seconds to load.

Before using a JavaScript framework, consider whether it's really necessary. In some cases, using small bits of JavaScript is more efficient than initiating calls to a framework.

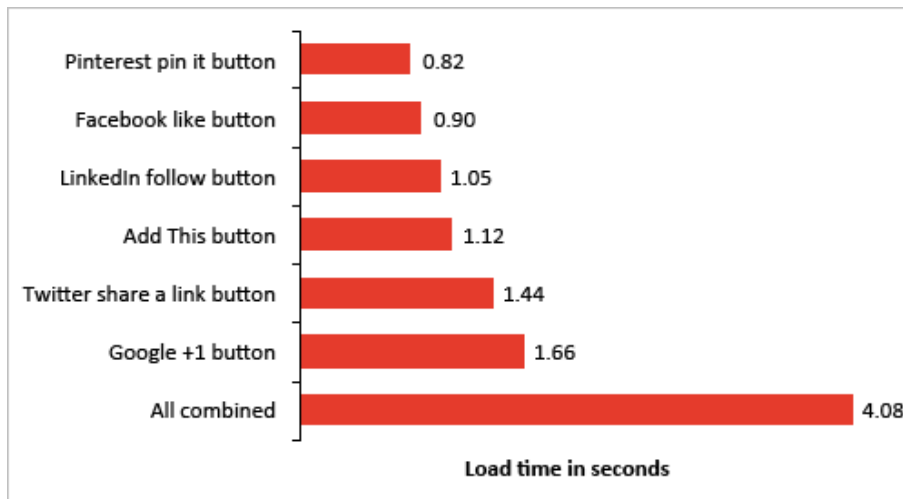
---

<sup>136</sup>. <http://www.starbucks.com>

<sup>137</sup>. <http://www.bbc.co.uk/mobile>

## AVOID WIDGETS

Widgets can have a surprisingly catastrophic impact on real loading time. To verify this, we created a series of simple HTML files, each containing the default embedding code for one widget. You can see in the results below how bad it gets. Note that this isn't a perfect test because these are controlled experiments in a simulated environment, but it's interesting nonetheless.



*Loading times for widgets on an iPhone 4, iOS 5.0.*

Combining them all on a single page results in a whopping four-second loading time for the widgets alone.

## Server-Side Techniques

In addition to optimizing the front end, server-side techniques can also be used to speed up loading times. These techniques are worth looking into, but won't be covered in this chapter:

- Cache HTTP redirects to speed up repeat visits;
- Merge HTTP redirect chains to reduce the number of redirects;
- Use HTTP compression to reduce the number of bytes (Gzip or DEFLATE).

## *Testing Performance On Mobile Devices*

Because of the unpredictability of mobile devices, testing performance on multiple devices is important. Here are some free performance-testing tools:

- Mobitest<sup>138</sup>, Akamai  
Generate waterfall charts and HAR files for the iPhone 4 iOS 7, iPad 2 iOS 7, iPad 3 iOS 7, Nexus S, and Motorola XOOM. Note that test results for the Nexus S were inconsistent with our own internal testing. Our server-access logs showed fewer HTTP requests when we tested on actual Android 2.x devices.
- “Network Panel<sup>139</sup>,” Chrome Developer Tools  
Generate waterfall charts and HAR files from the Chrome browser.

## *Conclusion*

To meet the high expectations of mobile users, you need a mobile-optimized website that loads in 4 seconds or less.

---

<sup>138</sup>. <http://mobitest.akamai.com/m/index.cgi>

<sup>139</sup>. <https://developers.google.com/chrome-developer-tools/docs/network>

The best way to hit that magic 4-second mark is to minimize the processing load on smartphones by reducing JavaScript and by optimizing the HTML, CSS and images.

Using the techniques above, you will be well on your way to building a snappy mobile Web experience! 🍷

# Creating High-Performance Mobile Websites

BY JAMES ROSEWELL 🍷

**Editor’s Note:** *This chapter features just one of the many solutions for creating high-performance mobile websites. We suggest that you review different approaches such as the previous chapter and the articles “Facing The Challenge: Building A Responsive Web Application”<sup>140</sup> and “Improve Mobile Support With Server-Side-Enhanced Responsive Design”<sup>141</sup> before choosing a particular solution.*

People start to lose interest in a website if they don’t get a response within three seconds<sup>142</sup>. Fulfilling this expectation for mobile phone users requires a different approach to usage analysis, design and testing.

This chapter expands on the techniques that Johan Johansson has explained before.

We’ll demonstrate methods to identify how people interact with a website differently on mobile devices, and the design decisions that can be made based on this understanding. Our objective is not only to improve Web performance but to increase the client’s return on investment.

---

<sup>140</sup>. <http://www.smashingmagazine.com/2013/06/12/building-a-responsive-web-application/>

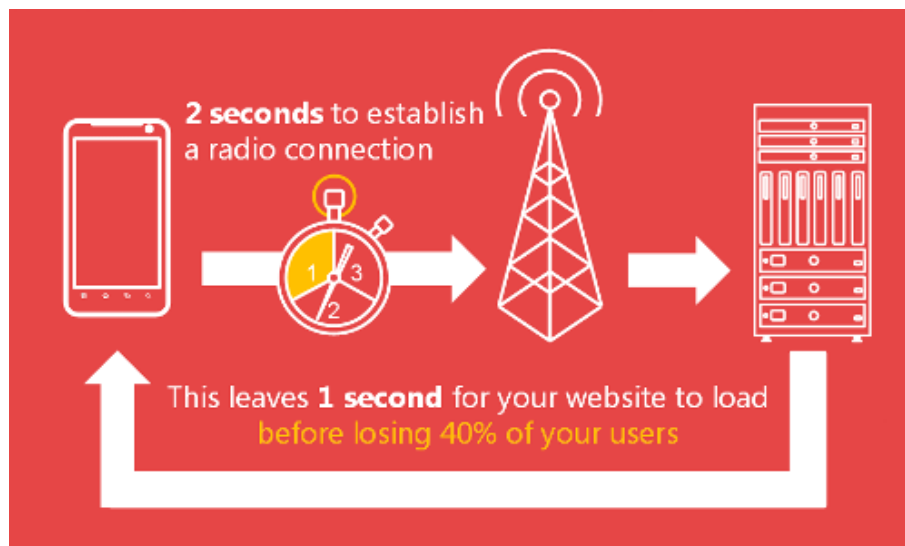
<sup>141</sup>. <http://www.smashingmagazine.com/2013/04/09/improve-mobile-support-with-server-side-enhanced-responsive-design/>

<sup>142</sup>. <http://51degrees.mobi/ressinfographic>

The techniques we'll demonstrate center on the two unique characteristics of mobile phones, which are not going to change any time soon: small batteries and small screens.

## SMALL BATTERIES

Mobile phones use radios for all communication, and they have little batteries that need to be carefully managed in order to avoid running out of power. As a result, radios are shut down very quickly when not in use, increasing the perceived time that a Web page takes to appear. 2G and 3G radios could require up to two seconds to establish an operational HTTP connection. If we accept that users start to lose interest after three seconds, then a website has only one second to respond. Think of this as the “golden second.”



*Maximizing the “golden second”<sup>143</sup>.*

---

<sup>143</sup>. <http://51degrees.mobi/ressinfographic>



## SMALL SCREENS

In the physical world, content is produced for billboards and magazines and customized to account for the size and viewing distance of the medium. In the digital world, a typical mid-range smartphone has a screen with around six square inches of real estate. A MacBook Pro with a 15-inch display will have over 100 square inches. Thus, not only can we optimize website performance by reducing the amount of content sent to phones, but we can optimize business processes to improve the return on investment for website owners.

The code examples in this chapter are provided in .NET. Where equivalents are possible in PHP, Java, C or Python, I've made them available in a companion article<sup>144</sup>. I'll explain why I've used .NET at the end of this chapter.

### *Maximizing The “Golden Second”*

Website designers and developers with high-bandwidth Wi-Fi and fixed-line connections often used to take bandwidth for granted. Responsive Web design (RWD) limits the creative process by forcing the same content, navigation and business processes to be presented on every device, irrespective of its physical capabilities.

Solutions to ensure that we can easily measure performance, monitor user behavior based on the characteristics of the device and optimize Web pages for low-band-

---

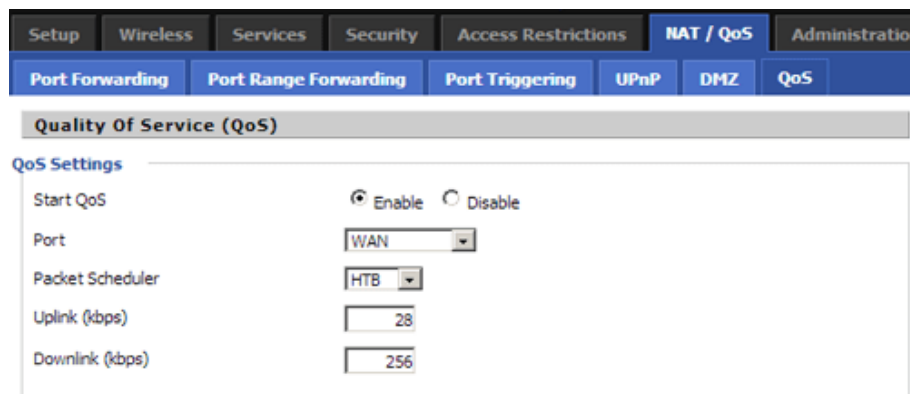
<sup>144</sup>. <http://51degrees.mobi/Blogs/tabid/212/EntryId/147/Understanding-Devices-That-Browse-Your-Website.aspx>

width devices are required to maximize the golden second.

## SIMULATING THE REAL WORLD

Essential for mobile Web performance testing is a method to simulate real-world mobile bandwidth conditions. Many wireless routers that cost less than \$100 support bandwidth limiting. This simply involves limiting the uplink and downlink bandwidth for LAN-side clients. If the router doesn't support this capability out of the box, then DD-WRT<sup>145</sup>, an open-source firmware upgrade, may be used to replace the default operating system on many popular routers to limit bandwidth.

I use a Linksys E3000 router modified with DD-WRT. The procedure to upgrade the router is pretty simple, and full instructions are available on the DD-WRT website.



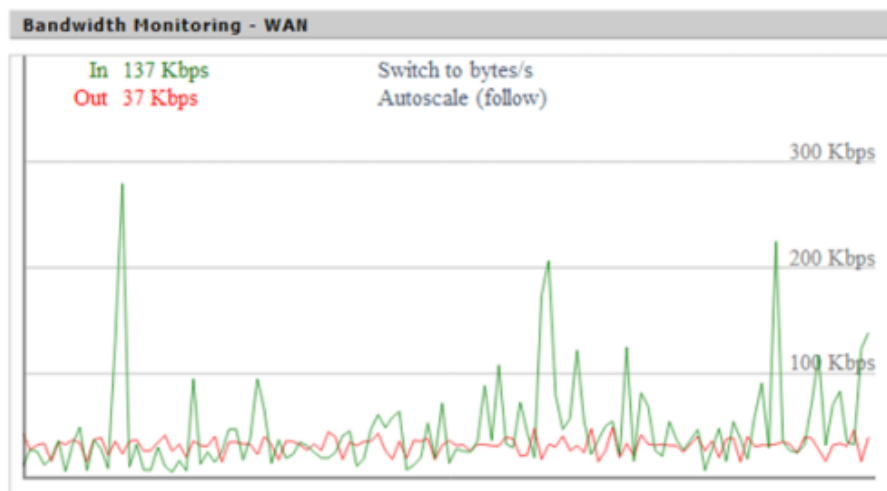
*Limiting bandwidth in the “Quality of Service” options.*

---

<sup>145</sup>. <http://www.dd-wrt.com/site/index>

Once DD-WRT is installed, go to the “QoS” (quality of service) menu, and enable bandwidth limiting. Then, set values for the uplink and downlink. I prefer 256 kbps for the downlink and 28 kbps for the uplink to simulate an average mobile bandwidth connection.

Now the bandwidth of any Wi-Fi or ethernet-cabled devices that are connected to the router will be artificially reduced. The actual bandwidth being used over time can also be monitored.



*Monitoring bandwidth using DD-WRT.*

While this approach doesn’t introduce random drop-outs, variable bandwidth conditions or the delays associated with radio wake-up, it is better than performing all of your testing on a fast low-latency broadband connection. When introduced at the beginning of the website development cycle, it’s an easy way to informally test performance during the development process and ensure that you don’t get any nasty surprises during formal testing.

## YOU CAN'T MANAGE WHAT YOU CAN'T MEASURE

Peter Drucker<sup>146</sup>, a management consultant, once famously said, “If you can’t measure something, you can’t manage it.”



*Average screen size growth over time.*

Continually monitoring the content that users view according to device characteristics (such as supported radios or physical screen size) will help you to identify the content and services that are more or less popular on mobile phones. Perhaps you will see no difference, but unless you measure it, there’s no way to know for sure.

---

<sup>146</sup>. [http://en.wikipedia.org/wiki/Peter\\_Drucker](http://en.wikipedia.org/wiki/Peter_Drucker)

## FEED ME NOW: AN EXAMPLE

A global fast-food franchise wanted to create a mobile-optimized version of its big-screen website. Before creating the first iteration of the mobile-optimized website, it performed analysis to determine which options on the big-screen website were being accessed by users on small-screen devices. The main menu, special offers and the store finder were the most popular, and so a mobile-optimized website was created that focused on these areas.

Work didn't stop there. Continued analysis revealed that the store finder was the most popular option. The mobile home page was altered again to focus on the store finder. Continued monitoring will show how many visitors choose other options, and the website will be continually refined to ensure that the most popular outcomes are catered to in the simplest possible way.

## BETTER LOGGING

Google Analytics provides some information about device model, but it lacks the detail we need to make informed decisions based on screen size and input method. Fortunately, a comprehensive device-detection repository (DDR) can be used to add this information to existing log files. The following code snippet can be added to a .NET website to obtain the screen's physical dimensions in inches and write the output to a simple CSV file.

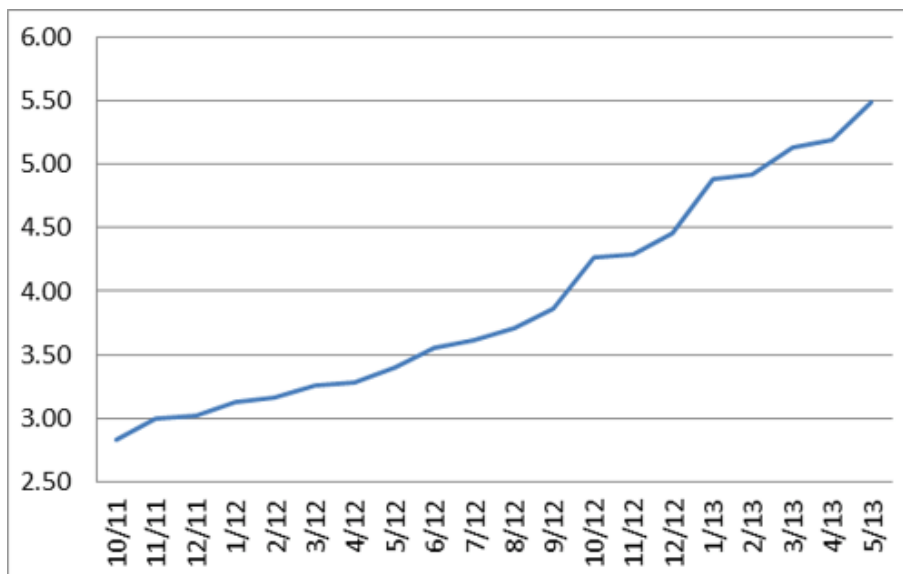
```
// Write a log file containing the current time, and the
// screen size of the requesting device in inches.
File.AppendAllText(
    Path.Combine(
```

```

AppDomain.CurrentDomain.BaseDirectory, String.Format(
    "App_Data\Simple_Log_{0:yyyyMMdd}.csv",
    DateTime.UtcNow)),
String.Format("{0:s},{1},{2},{3}\r\n",
    DateTime.UtcNow,
    Request.Path,
    Request.Browser["ScreenInchesWidth"],
    Request.Browser["ScreenInchesHeight"]));

```

The first column is the date and time that the request was processed. The second is the page being requested. The final two columns are the width and height in inches. Once sufficient data has been captured, the average screen size in square inches can be calculated and plotted on a chart, similar to the following:



*Comparison of the average sizes of device screens over 20 months.*

The analysis could be narrowed down to individual pages. Other characteristics about device, operating system and browser may be added as columns.

Similar code could be used with PHP, Java, Python and other environments.

## EXISTING LOG FILES

Sometimes, existing Web pages can't be altered in the way shown. In these situations, a DDR may be used to perform offline analysis of log files containing user agents. The following .NET code is a functional command-line program that will parse a space-separated log file and calculate the average screen size in square inches for the requests it represents. The first argument is the log file's location, the second is the index of the **UserAgent** column within the log file.

```
using System;
using FiftyOne.Foundation.Mobile.Detection.Binary;
using System.IO;

namespace ConsoleApplication
{
    class Program
    {
        static void Main(string[] args)
        {
            // The number of devices read from the log file.
            int count = 0;
```

```

// The column in the input file the user agent is
// held in.
int column = int.Parse(args[1]);

// Screen dimension variables.
double total = 0, width, height, squareInches;

// Create a provider to determine the device
// capabilities.
var provider =
Reader.Create("51Degrees.mobi.dat");

// Read each line of the log file provided in
// argument 0.
// Assume the value at column 8 is the UserAgent
// string.
using (var reader =
File.OpenText(args[0]))
{
    while(reader.EndOfStream == false)
    {
        var values =
        reader.ReadLine().Split(new[] { ' ' });
        if (values.Length >= column)
        {
            // Get the device information based
            // on the UserAgent.
            var device = provider.GetDeviceInfo(
                values[column - 1].Replace("+",
                " "));
        }
    }
}

```



```

        if (device != null)
        {
            // Determine the screen
            // dimensions in inches.
            double.TryParse(
                device.GetFirstPropertyValue(
                    "ScreenInchesWidth"),
                out width);
            double.TryParse(
                device.GetFirstPropertyValue(
                    "ScreenInchesHeight"),
                out height);
            squareInches = width * height;
            // If valid values are available
            // (not a desktop/laptop) then
            // add the values to the results.
            if (squareInches > 0)
            {
                total += squareInches;
                count++;
            }
        }
    }
}

Console.WriteLine(
    "Average screen size '{0:#.00}'
    square inches from '{1}' devices",
    total / count,

```

```

        count);
    Console.ReadKey();
}
}
}

```

Analyzing log files is less accurate because HTTP headers other than **User-Agent** affect the detection's results. This is especially true with Opera Mini and Opera Mobile browsers, in which a second HTTP header, named **Device-Stock-UA**<sup>147</sup> is used to provide information about physical hardware not available in the standard **User-Agent**.

## WHY MONITOR?

Monitoring enables us to remove unpopular content from major landing pages, thus improving the performance of content that is more important or relevant. The removed content should still be available via second-level pages — just not placed on landing pages, where they would eat up valuable bandwidth and slow down performance.

So, how do we create a separate mobile website optimized for performance?

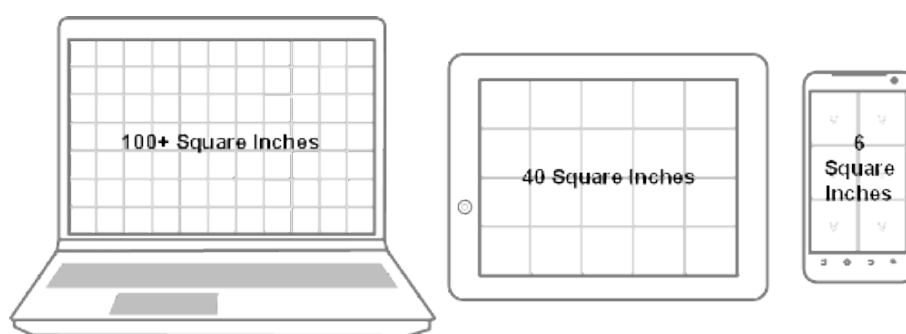
## DIVIDE AND CONQUER

I understand why RWD makes a lot of sense from the perspective of user interface design. It's great in situa-

---

<sup>147</sup>. <http://my.opera.com/ODIN/blog/2012/10/08/introducing-device-stock-ua>

tions in which content, navigation and business-process requirements are identical between 6-square-inch screens and 100-square-inch screens and only the layout needs to be altered.



*Average device screen size.*

However, having a separate mobile website makes a lot of sense when the conditions above aren't true or when performance is critical.

Separate mobile websites are often implemented in a way that delivers a poor user experience. Google is now shining a light<sup>148</sup> on these common issues by penalizing websites with lower search engine rankings. Mistakes include sending every desktop page to a single mobile home page, redirecting to application download pages, preventing the user from accessing the big screen website, and treating all devices with a particular operating system in the same manner.

These poor implementations have given the concept a bad reputation. Here's how to do it simply and properly.

---

<sup>148</sup>. [http://googlewebmastercentral.blogspot.co.uk/2013/06/changes-in-rankings-of-smartphone\\_11.html](http://googlewebmastercentral.blogspot.co.uk/2013/06/changes-in-rankings-of-smartphone_11.html)

The following .NET `web.config` section will redirect the first request from a smartphone to an equivalent page on the “Smartphone” section of the website. Importantly, the query string and page name are retained across the redirection.

```
<redirect firstRequestOnly="true"
  mobileHomePageUrl="~/Mobile/Default.aspx"
  timeout="20"
  devicesFile="~/App_Data/Devices.dat"
  mobilePagesRegex="/(Mobile|Smartphone)/" >
  <locations>
    <!--Send smartphones to an equivalent version
    of the original page, preserving the
    page name and query string.-->
    <location name="smartphone" url="~/Smartphone/{0}"
      matchExpression="(?!<=^w+://.+/.).+>
      <add property="IsSmartphone"
        matchExpression="true"/>
    </location>
  </locations>
</redirect>
```

In most situations, when redirected to alternative pages, users should be able to return to the original page if they wish; perhaps they’re more familiar with the big-screen version of the website. The `firstRequestOnly` attribute ensures that only the first request from the device is redirected. The `devicesFile` attribute is used to track devices on which cookies aren’t supported. The `timeout` attribute

controls how long the device is remembered (for the purpose of redirection).

The redirection system also has to know which pages are designed for which type of device. The **mobilePages-Regex** attribute is applied to requested URLs. If there is a match, then the page won't be eligible for redirection. This prevents cases of infinite redirections.

The **locations** element allows for different locations and associated rules to be configured. The example inserts the folder **Smartphone** into the original URL. The query string and other URL information are retained across the redirection. All information that affects the context of the request must be transferred in order for the user to receive the content they are expecting.

This simple approach enables a search engine-friendly, Google-compliant, mobile phone-optimized website to be delivered, with a good user experience and superior performance. Essential to the process is a DDR that provides information about the device quickly, consistently and accurately. Users who change their mobile phone's browser settings to surf in desktop mode will be respected, and the redirection will not occur.

## BEWARE OF CLOUDS

Cloud services are a popular way to easily add features to a website. But they bring a performance penalty by calling out over the Internet. Ignoring processing time, we've observed an average 200-millisecond delay<sup>149</sup> with data transmission from Amazon Web Service-hosted cloud services.

200 milliseconds is 20% of the golden second. Therefore, consider carefully where you use cloud services, ensuring they're called asynchronously to enable other processing to continue while waiting for the response. They should be avoided for critical path activity, such as determining information about the requesting device.

## *Squeezing Content*

After video, images, CSS and HTML make up the bulk of Web traffic. We need methods of optimizing them all. Video is an article on its own and will have to wait for another day.

### **IMAGES**

A popular solution is to provide three versions of the same image, and select the one that is best for the requesting device using JavaScript or CSS3 when the browser renders the page. This is a great start, but managing different versions of the same image is a pain; the image is never ideally optimized, and the method puts the burden of resizing onto the mobile device's limited CPU and battery.

There is a better way, using an image optimizer. There are many great options out there; if you decide to use our very own image optimizer, you can add it to an ASP.NET website via the Visual Studio IDE. The following configuration will be added automatically to the **web.config**.

---

149. <http://51degrees.mobi/Blogs/tabid/212/EntryId/99/Is-Cloud-Mobile-Detection-Compromising-Your-Mobile-Web-Experience.aspx>

```
<handlers>
  <add name="Image" verb="GET" path="P.axd"
    type="FiftyOne.Framework.Image.ImageHandler,
    FiftyOne.Framework" />
</handlers>
```

The handler tells Internet information services (IIS) that the image handler should process any **GET** requests for the resource **P.axd**.

Once enabled in **web.config**, the following ASP.NET code will use the image optimizer to define an image element with three possible sources – being 240, 480 and 640 pixels wide, respectively.

```
<mob:Image runat="server" ID="ImageBanner"
CalculateSizeMode="ClientWidth" Style="clear: both; width:
100%">
  <mob:AltImage ImageUrl="~/Images/Landscape240.png" />
  <mob:AltImage ImageUrl="~/Images/Landscape480.png" />
  <mob:AltImage ImageUrl="~/Images/Landscape640.png" />
</mob:Image>
```

When the image is initially displayed, the server will send a white 1 × 1-pixel GIF to appear in place of the image. This is the resulting HTML:

```

```

Once the page has loaded, JavaScript is used to work out the exact dimensions required by the image and request a precisely sized image from the server. After JavaScript processing, the HTML above will be transformed to this:

```

```

The image handler referenced in `web.config` correlates the `I` query string parameter with the sources of the image, so that the best image can be used as the starting point for resizing on the server. The `w` query string parameter specifies the width of the image required. Multiple images don't need to be provided; a single image will work almost as well.

This approach is easy to implement, and the result is a precisely sized image, which reduces bandwidth consumption, mobile phone CPU cycles and power consumption.

## HTML

The full Oxford English Dictionary contains 171,476 words. If a computer were to represent each word as a unique binary number, rather than letters in an alphabet, then 18 bits (or 3 bytes, if rounded up) would be required. This technique is why compression algorithms are so effective.

However, HTML is not very efficient because it's full of words for elements, IDs, classes, styles and JavaScript, without even considering the human-readable words. Compression reduces this, but it remains an overhead. This is why popular libraries have minified versions that appear almost unreadable to humans.

Some of these markup-related words can also be minimized before being sent to the browser by the server, without losing any of their meaning. Taking the image example shown earlier, the standard HTML ID attribute



of the image element in ASP.NET would be **ImageBanner**.

```
<mob:Image runat="server" ID="ImageBanner"
CalculateSizeMode="ClientWidth" Style="clear: both; width:
100%">
```

However, the code sent to the browser would use just **B**. For a single element, the performance improvement is negligible, but on a complex page with hundreds of elements, the page will transfer more quickly and the browser will be able to process everything that much faster.

## INCLUDES

Something else is slightly peculiar about the resulting HTML from the image example.

```

```

The ASP.NET code includes a style attribute that is missing, and there isn't a class attribute for the **img** element. So, how is the style being applied?

The server-side-minimizing process will identify style information and create a CSS include for the page, thus reducing the HTML. If the HTML changes, then the style information will already have been cached in the browser and will not need to be downloaded again. The CSS snippet looks like this:

```
#B{clear:both;width:100%;}
```

If many elements share the same styling, then their IDs will be added to the CSS and they'll share the same information.

Style information can also be shared across elements and pages using a server-side style element. The following code extends the previous image example to demonstrate a shared style element.

```
<mob:Style runat="server" ID="StyleBanner">
    <mob:Filter Style="clear: both; width: 100%"/>
</mob:Style>

<mob:Image runat="server" ID="ImageBanner"
CalculateSizeMode="ClientWidth" StyleID="StyleBanner">
```

The elements can be further extended to apply different styling based on the capabilities of the device and to optimize style sheets across multiple pages.

This technique will always ensure that only the required CSS is transferred, thus improving performance over subsequent requests to the same page, particularly where there are only minor differences in HTML content.

## WHY .NET?

The techniques and code examples shown for image optimization and dynamic minification of HTML and CSS content rely on content being altered after the page has been rendered but before transmission to the browser by the server. Such preprocessing techniques are relatively easy to implement in architectures such as ASP.NET Web forms.

However, they are a lot more complex to implement in script-based architectures such as PHP. For this reason, the examples in this chapter are provided in .NET for con-

sistency. Where I've been able to apply the techniques to other languages, the example code is available in a companion blog<sup>150</sup>.

## Examples

Public Health Foundation Enterprises implemented the techniques<sup>151</sup> shown in this chapter and experienced a 23% increase in successful outcomes during the first week.

Other performance-aware websites — including 24.com<sup>152</sup> (media), ServiceTick<sup>153</sup> (analysis), LettingWeb<sup>154</sup> (property), AdSupply<sup>155</sup> (advertising) and Kitsap Credit Union<sup>156</sup> (finance) — are all optimizing for mobile using some or all of techniques covered in this chapter.

## Summary

We need to consider the return on investment for a website's owner in order to truly optimize performance. Monitoring differences in the characteristics of devices is the essential starting point.

We can then deploy solutions such as using separate mobile websites to split up or change the focus of con-

- 
- <sup>150</sup>. <http://51degrees.mobi/Blogs/tabid/212/EntryId/147/Understanding-Devices-That-Browse-Your-Website.aspx>  
<sup>151</sup>. <http://51degrees.mobi/Products/CaseStudies/PHFEWIC.aspx>  
<sup>152</sup>. <http://51degrees.com/Resources/Case-Studies/24com>  
<sup>153</sup>. <http://51degrees.com/Resources/Case-Studies/ServiceTick>  
<sup>154</sup>. <http://51degrees.com/Resources/Case-Studies/Lettingweb>  
<sup>155</sup>. <http://51degrees.com/Resources/Case-Studies/AdSupply>  
<sup>156</sup>. <http://51degrees.com/Resources/Case-Studies/KITSAP>

tent. And we can squeeze maximum performance out of mobile phones by minifying images and HTML, removing jQuery, questioning when to use RWD alone, and other techniques. Of course, established techniques are critical, too, such as configuring caching directives and compressing content.

Tweaking our development environment to simulate real-world conditions will also yield a greater understanding of performance throughout the development process.

## OPTIMIZE NOW

To get you thinking even more about performance, I've set up a competition to find the world's heaviest website<sup>157</sup>. (*Editor's Note:* The competition is closed now.) Find a Web page that performs poorly on a mobile phone and submit it to the competition. We'll work out the page's weight, and if it's the heaviest, you'll win \$1000. Meanwhile, implement the techniques covered in this and other great Smashing Magazine articles to ensure that your website doesn't top the list when we weigh in on performance!

There's never been a better time to improve your website's performance. 🐼

---

<sup>157</sup>. <http://51degrees.mobi/Competitions/HeaviestWebSite2013.aspx>

# Don't Get Crushed By The Load: Optimization Techniques And Strategies

BY BOBBY PEARSON 🍷

Despite improvements in broadband<sup>158</sup> and wireless Internet<sup>159</sup>, load is in many ways more of an issue now than it was five years ago. The proliferation of mobile devices, increased user expectations<sup>160</sup>, and the very real risks of losing customers<sup>161</sup> and dropping in search result rankings<sup>162</sup> have laid a heavy burden on developers to optimize loading time at all costs.

In building websites primarily for the desktop environment, the Web development community previously didn't spend much time concerning itself with load issues. Selecting the proper image formats and saving our JPEGs for the Web was about as far as many of us would go. On the whole, our hardware and software tools are forgiving enough to accommodate sloppy code. Our production environments can handle thousands of visitors

- 
- <sup>158</sup>. <http://techcrunch.com/2012/08/09/akamai-global-average-broadband-speeds-up-by-25-u-s-up-29-to-6-7-mbps/>
- <sup>159</sup>. <http://www.statista.com/topics/779/mobile-internet/chart/1009/mobile-internet-traffic-growth/>
- <sup>160</sup>. <http://www.nytimes.com/2012/03/01/technology/impatient-web-users-flee-slow-loading-sites.html>
- <sup>161</sup>. <http://www.topfloortech.com/blog/2012/02/10/why-slow-loading-websites-lose-customers/>
- <sup>162</sup>. <http://www.quicksprout.com/2012/12/10/how-load-time-affects-google-rankings/>

per day, and our clients tend to have predictably manageable traffic.

For all of these reasons and more, Web developers aren't conditioned to think very hard about the unique load requirements of their clients' websites. Predictability and complacency can leave our websites vulnerable to traffic spikes, glacial page loading and even downtime. We need to include a specification for load requirements as a regular checklist item when bidding and planning Web work.

## *Learn To Love The Content Delivery Network*

Content delivery networks (CDNs) are everywhere. And they are our friends. A CDN is a network of servers arrayed across the Internet. These servers work in tandem to serve website content scripts, images, fonts, audio, video and other files – in a distributed fashion. Using a CDN to serve content, your website's files are pushed out to the edges of the network, several hops and several hundred milliseconds closer to your users.

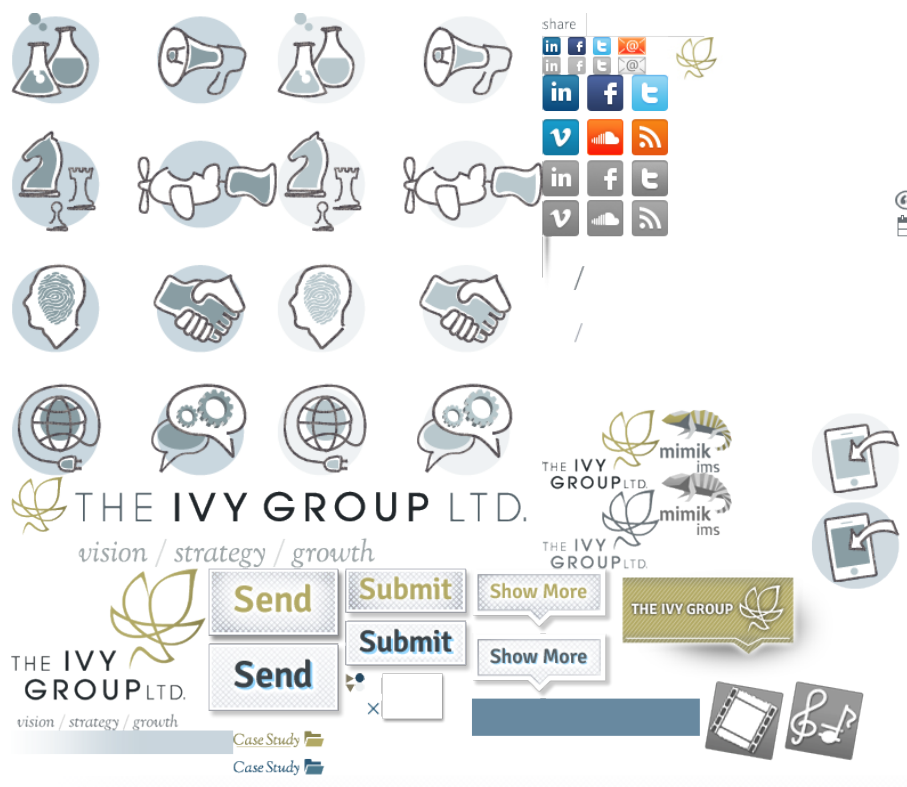
I highly recommend storing your website's graphic elements on a CDN. Delivering that 300 KB background image and multi-image sprite<sup>163</sup> on a CDN will dramatically improve your page's speed and decrease the load on your server. When my company started its website's mobile-optimized redesign, we ran Google's PageSpeed

---

<sup>163</sup>. <http://webdesign.tutsplus.com/tutorials/htmlcss-tutorials/css-sprite-sheets-best-practices-tools-and-helpful-applications/>

Insights<sup>164</sup> tool and found numerous problems – the scores ranged from 70/100 down to 50/100.

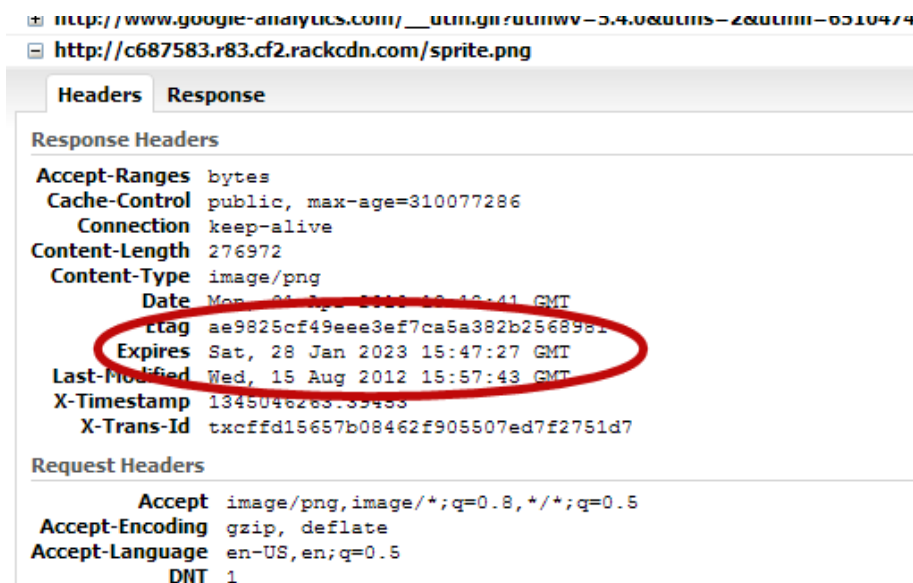
One of the best decisions we made was to combine more than 40 of our images into a single sprite and serve that sprite over our hosting provider's cloud files system. Rather than loading 40+ individual image files as needed, we have the user load one 275 KB file that is not set to expire for 10 years. Unless we manually update and purge the file on our CDN, users will be able to load a cached version of it from our CDN's edge nodes (or from their own browser cache) until 2023. Hopefully, bandwidth won't be as big of an issue by then!



The 40-image sprite used by The Ivy Group<sup>165</sup>.

<sup>164</sup>. <https://developers.google.com/speed/pagespeed/insights>

By using the CSS `background-position` property<sup>166</sup> to display most images as sprites from a master image, we've limited the number of HTTP requests. Each `div` that contains a sprite has a defined height and width in the CSS; so, until the master image is loaded, the user is presented with a blank box.



The expiration header for our sprite is set for almost 10 years in the future, ensuring long-term caching on browsers and CDN edge nodes.

Find out from your hosting provider whether it offers CDN services with its hosting packages. Alternatively, providers such as Rackspace<sup>167</sup>, BitGravity<sup>168</sup> and EdgeCast<sup>169</sup> are CDN specialists and can serve as an auxiliary

<sup>165</sup>. <http://ivygroup.com/>

<sup>166</sup>. <http://www.smashingmagazine.com/2009/04/27/the-mystery-of-css-sprites-techniques-tools-and-tutorials/>

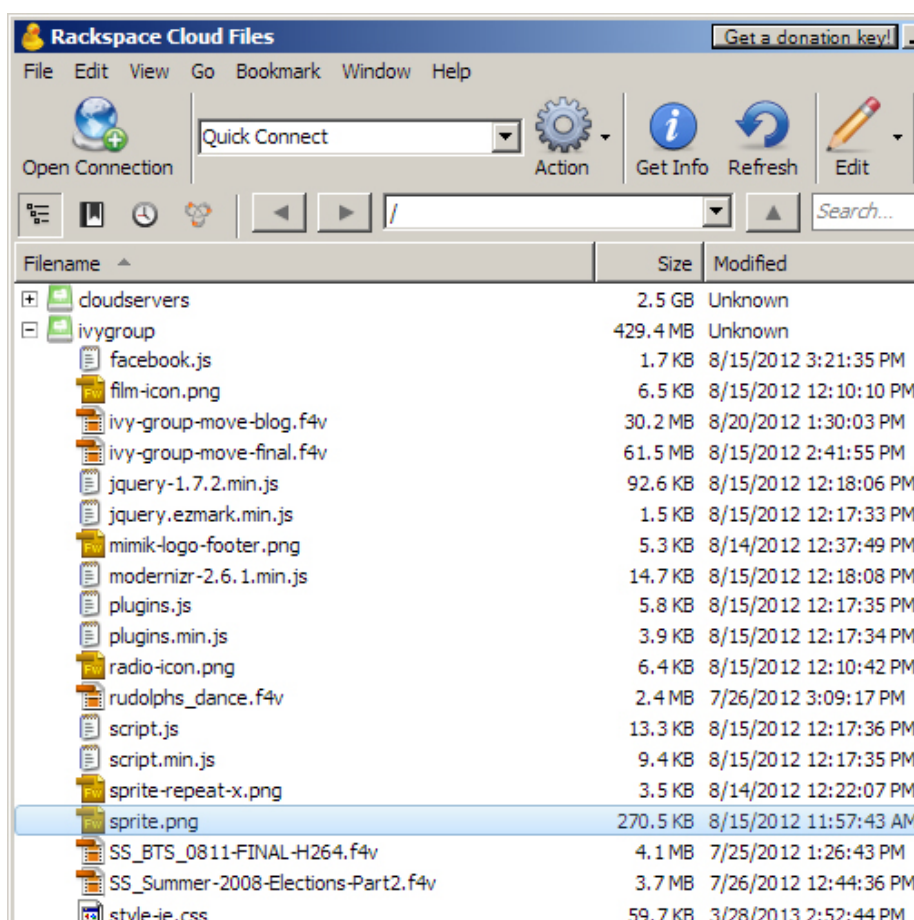
<sup>167</sup>. <http://www.rackspace.com/cloud/files>

<sup>168</sup>. <http://www.bitgravity.com/>

<sup>169</sup>. <http://www.edgecast.com/>



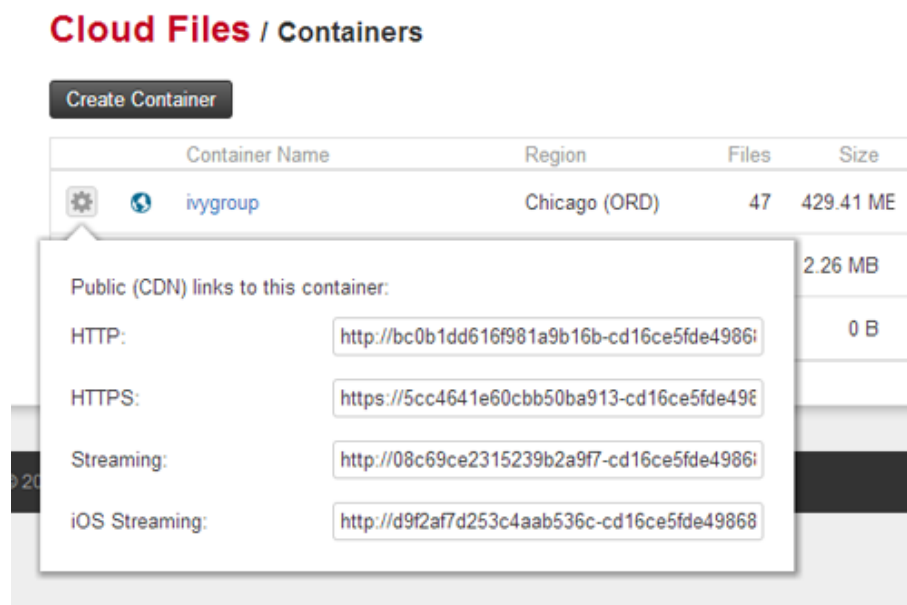
service for your existing hosting solution. Rackspace and BitGravity are both resellers of the Akamai CDN, while EdgeCast runs its own network. Be sure to ask how the CDN will bill you. When I was pricing CDNs for a large file-delivery project, I found some companies will charge you only for downloads from the core of their network, not from the edge nodes, while others charge you for all downloads, regardless of which level of the network responds to each user request.



A collection of files in the **ivygroup** CDN container, as viewed in Cyberduck.

To host your website's resource files on a CDN, first obtain an API user name and key from your provider. Use a cloud-file browser such as [Cyberduck](http://cyberduck.ch/)<sup>170</sup> to create a CDN “container” (essentially, a directory) for each project or client, and upload the files just as you would with an FTP connection.

Then, copy the container's public HTTP or HTTPS path to your clipboard.



*The HTTP, HTTPS, Streaming and iOS Streaming public links to the ivygroup CDN container.*

Then, simply edit your HTML or CSS code, changing the file paths to reflect the public CDN links. Below is a sample block of CSS showing the HTTP public links to browser-specific style sheets:

---

<sup>170</sup>. <http://cyberduck.ch/>

```

<!--[if !(IE)]><!--> <link href="http://bc0b1dd616f981a9b16b
-cd16ce5fde498683e20aaecc086aa721.r81.cf2.rackcdn.com/
style.min.css?v=1.31" rel="stylesheet" /><!--<![endif]-->
<!--[if gte IE 9]>
<link rel="stylesheet" href="http://bc0b1dd616f981a9b16b
-cd16ce5fde498683e20aaecc086aa721.r81.cf2.rackcdn.com/
style.min.css?v=1.31">
<![endif]-->
<!--[if lt IE 9]>
<link rel="stylesheet" type="text/css" media="all"
href="http://bc0b1dd616f981a9b16b-cd16ce5fde498683e20aaecc
086aa721.r81.cf2.rackcdn.com/style-ie.css?v=1.10"/>
<![endif]-->

```

And here is a sample block of CSS showing how to style a Twitter button (with a hover effect) using a sprite hosted on a CDN:

```

.social-media-block a {
    background: url("http://bc0b1dd616f981a9b16b-cd16ce5fde49
8683e20aaecc086aa721.r81.cf2.rackcdn.com/sprite.png")
    no-repeat transparent;
    float: left;
    height: 32px;
    margin: 0 7px 7px 0;
    padding: 0;
    width: 32px;
}

.social-media-block a#twitter {
    background-position: 252px 667px;
}

```

```

}

.social-media-block a#twitter:hover {
    background-position: 252px 742px;
}

```

## Single Vs. Multiple Resource Domains

Linking to resource files on a CDN brings up an interesting optimization issue. Web browsers limit the number of files<sup>171</sup> that may be loaded in parallel from a single host. You'd think, then, that dispersing all of your resource files across many different host servers would be optimal (for example, `content1.mydomain.com`, `content2.mydomain.com`, etc.) — this is known as domain sharding<sup>172</sup>. However, there is a tradeoff, because each host name requires a new DNS lookup to find the host's IP address, which imposes a cost in loading time. Mobile browsers that connect via the cellular network tend to experience longer lookup times than browsers that connect via traditional cable and DSL modems. It's a good idea to evaluate your website once all content and resource files are in place and to experiment with multiple domains.

Mobify conducted a test last year that indicated that domain sharding was “at best neutral and, in most cases, detrimental<sup>173</sup>” to mobile browsers, but feel free to con-

---

<sup>171</sup>. <http://gtmetrix.com/parallelize-downloads-across-hostnames-implementation-guide.html>  
<sup>172</sup>. <http://gtmetrix.com/parallelize-downloads-across-hostnames.html>  
<sup>173</sup>. <http://www.mobify.com/blog/domain-sharding-bad-news-mobile-performance/>

duct your own testing. At the very least, minimize the total number of files loaded per page, avoid linking to URLs that serve only as redirects to other domains, and use a standard host name convention (i.e. either [www.mydomain.com/file.jpg](http://www.mydomain.com/file.jpg) or [mydomain.com/file.jpg](http://mydomain.com/file.jpg), not both).

## *Identify And Anticipate High-Traffic Periods*

Suppose you develop a website for a high-profile event or for an organization whose business is concentrated around a few dates every year. Systems that work fine during the slow periods need to be tested with load in mind.

This is particularly important to administrators who use shared hosting and have to impose traffic caps on their clients' websites to protect the server. You might set a traffic cap during the slow period that shuts down the website during a natural high-volume spike, such as one towards the end of a registration period for an event. Be sure to set a generously high traffic cap on any website for which you expect seasonal or irregular traffic patterns, or, better yet, host in a cloud environment that is scalable to meet any traffic demand.

I still burn with shame when remembering how I mistakenly set a hard, rather than a soft, traffic cap on a charity fundraiser website during the "off-season," only to have the website shut down on the weekend of the event itself. The website had been bouncing along at a stable traffic rate (about half a gigabyte per month) for months,

so I set the cap to 2 GB per month and billed the client accordingly (admittedly a low figure, but our hosting model is based on customization and technical support and not on unlimited, unsupported space on a server farm). The website drew almost 2 GB of traffic on the Saturday of the event alone and went down at the worst possible time.

Don't let this happen to you! My company learned its lesson: We now use only soft traffic caps as notifications of abnormal activity and investigate any notifications on the same day. Speak with your hosting provider to determine its preferred means of managing traffic spikes (see the following section on elasticity and load balancing), and make sure that you can stand behind whatever package of products and services you offer to clients.

## *Scalable, Elastic Architecture On The Cloud*

Beyond server- and client-side software, there is a wealth of options for how to structure and optimize your website's hardware infrastructure. Two concepts here are scalability and elasticity, covered in an educational article<sup>174</sup> by Cloud IQ's Guy Fardone. Scalability is the capacity to increase the load that your website will normally expect: the number of users supported by the system, the maximum size of all uploaded files, etc. A scalable solution is able to be multiplied by a factor of X (for example,

---

<sup>174</sup>. <http://blog.evolveip.net/index.php/2012/05/24/cloud-elasticity-and-cloud-scalability-are-not-the-same-thing-2/>

“Create three fresh new copies of the system for three new clients,” or “The client has expanded their user base to include all employees beyond the 10-user pilot group; create 1,000 new user accounts each with a 1 GB file store”). Elasticity is the capacity of the website’s infrastructure to expand on demand during load spikes.

Many high-profile websites have implemented a multi-server solution with load balancing. A single load balancer accepts all client traffic and directs each client to one of N available Web servers, which in turn are able to share state information through either a centralized or asynchronous state-management scheme. MSDN has a helpful article about load-balancing concepts<sup>175</sup> that is good for beginners (scroll down to figures 2 and 3). Speak with your cloud hosting provider to see what it can provide you.

## *Plan For Load Testing*

Last year, my company finished the front-end work for a digitized collection of documents related to the founding fathers of the United States, overseen and funded by the US National Archives. Our client, the University of Virginia’s University Press, wisely budgeted several months to load-test the database and document-delivery system in conjunction with the National Archives’ technical team.

This made us realize that small Web development businesses and freelancers, too, often take load testing for

---

<sup>175</sup>. <http://msdn.microsoft.com/en-us/library/ff648960.aspx>

granted. If a page loads within a few seconds while we're developing, what's the problem? By disregarding loading issues, the developer, first, misses the opportunity to add value to the delivered product and, secondly, allows a potential risk to go unmanaged.

Several load-testing services are out there: Load Impact<sup>176</sup> and Blitz<sup>177</sup> offer free trials and are worth investigating. If you have the technical ability and permission level, you could also try installing a server-side open-source load-testing tool such as JMeter<sup>178</sup> or ab<sup>179</sup> from Apache. JMeter can test a wide variety of server and service types, while ab specializes in HTTP benchmarking. A reliable load tester that you would be comfortable operating and in whose results you can be confident would make a useful addition to your toolbox.

Below is a screenshot of a Blitz load test on my company website's news page. Although the page has a very high page-speed score of 95 — due to optimized images, browser caching, minified CSS and JavaScript files, deferred loading and a few other tricks (more on those later!) — it could use some database and memory optimization. A single user will start to see formatted content displayed on their browser within 0.2 seconds, and the full page will load within 1.2 seconds. However, the page runs the risk of throwing errors and timeouts when the number of concurrent users hits 34.

---

<sup>176</sup>. <http://loadimpact.com/>

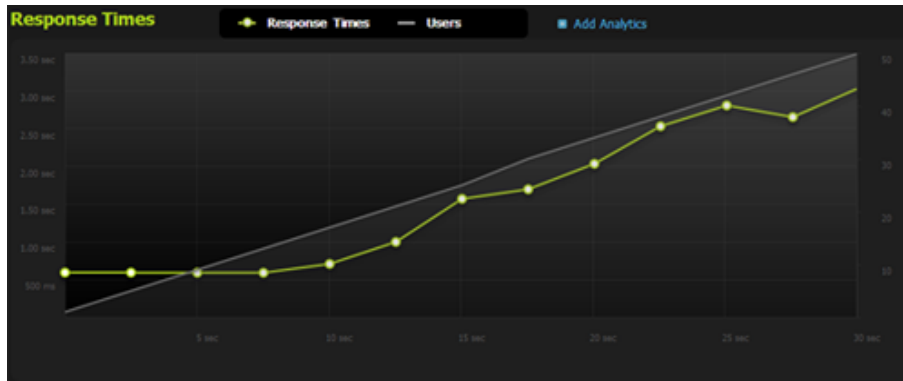
<sup>177</sup>. <http://blitz.io/>

<sup>178</sup>. <http://jmeter.apache.org/>

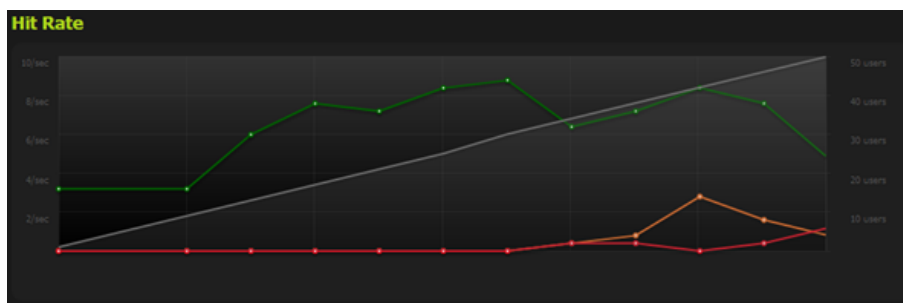
<sup>179</sup>. <http://httpd.apache.org/docs/2.2/programs/ab.html>



## PRE-OPTIMIZATION: 6 ERRORS, 16 TIMEOUTS, MAX 3.0-SECOND RESPONSE TIME



*As the number of simultaneous users increases from 0 to 50 over a 30-second period, the response time climbs from 600 milliseconds to just over 3 seconds.*



*Green denotes hits; red, errors; and orange, timeouts. The website maxes out at 9 users per second around the 18-second mark. At 25 seconds, 3 users per second are experiencing timeouts. At 30 seconds, over 1 user per second is receiving an error message.*

I took steps to remedy this by increasing the MySQL connection pool and by implementing in-memory caching via memcached. (Note that these are advanced server-administration tasks and should not be attempted without expert assistance.) Increasing our maximum number of allowed MySQL connections from 100 to 200 failed to

change our load-testing results — your mileage may vary. However, I did have success when I decided to...

## *Use Memory Caching*

Memcached is a fantastic extension that can be used to selectively store and retrieve query results from memory and to avoid making repetitive database calls, file-read operations and other server-side calculations. Other memory caching extensions are available that are worth comparing, but they all have some sort of key-value pair architecture for quick read/write access in memory and a fixed time-to-live so that data won't cache persistently and result in an overflow.

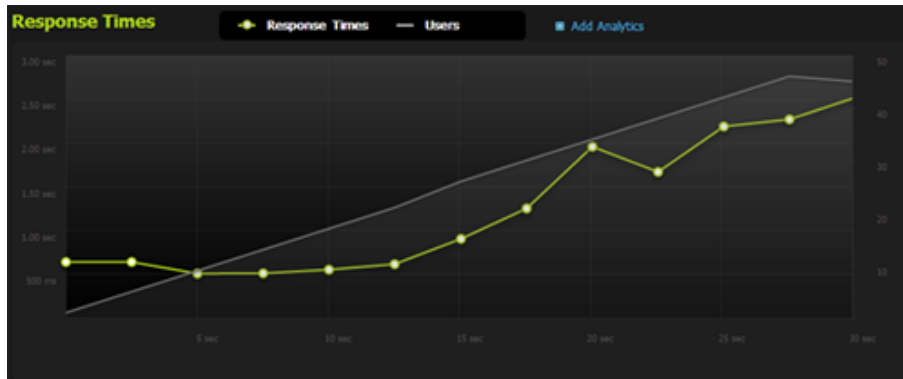
Script Tutorials has two sets of sample code that demonstrate “How to Use APC Caching<sup>180</sup>” and “How to Use Memcache in PHP<sup>181</sup>.” I selected memcached and successfully improved our website's performance — adding a single `memcache()` call in one of my most-used database functions decreased our total errors during a 30-second rush from 6 to 1 and our total timeouts from 16 to 7.

---

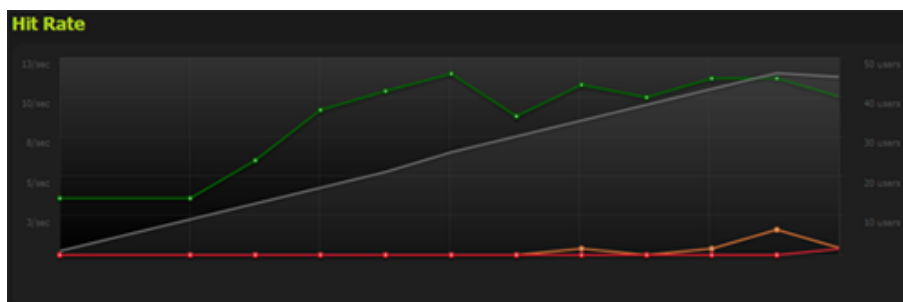
<sup>180</sup>. <http://www.script-tutorials.com/how-to-use-apc-caching-with-php/>

<sup>181</sup>. <http://www.script-tutorials.com/how-to-use-memcache-with-php/>

## POST-OPTIMIZATION: 1 ERRORS, 7 TIMEOUTS, MAX 2.5-SECOND RESPONSE TIME



After applying memcached to one database function, the maximum response time at peak load fell from 3 to 2.5 seconds. The website performs measurably better during the 0- to 20-second period (between 1 and 35 simultaneous users). Note that the y-axis scale is not equal to the y-axis of the previous “Response Time” graph.



Green denotes hits; red, errors; and orange, timeouts. After applying memcached to one database function, the website experienced a marked decrease in errors and timeouts during high usage. Further use of memcached is recommended. Note that the y-axis scale is not equal to the y-axis of the previous “Hit Rate” graph.

## MYSQL QUERY CACHE

Memcached isn’t the only caching tool at your disposal. MySQL enables you to access query results directly from memory using Query Cache<sup>182</sup>. It has both advantages

and disadvantages when compared to memcached; the two should not be treated as interchangeable. MySQL Query Cache is ideal for caching data that is seldom changed, as Rackspace explains<sup>183</sup>. Rackspace's key takeaway is this:

*When you use a MySQL database, each time you INSERT, UPDATE, or DELETE a row from the MySQL database, the entire query cache for that table is invalidated. That means that the next request for every single session will be a cache miss, and must access the data from disk on the database server.*

Memcached treats every SQL query and result as a unique key-value pair, so an entire table's worth of **SELECT** queries will never be invalidated all at once if anything is changed. MySQL Query Cache recognizes that any change to any row in a table might affect any further queries on that table. It's up to you, with your knowledge of your application, to determine whether one or both of these tools are worth implementing.

Test different values of the **query\_cache\_size** parameter in MySQL Query Cache and of both the cache size and timeout limit in memcached. There is no "correct" value for any of these parameters. It's all based on your hosting environment, application structure and expected load.

---

<sup>182</sup>. <http://www.docplanet.org/mysql/mysql-query-cache-in-depth/>

<sup>183</sup>. <http://www.rackspace.com/blog/memcached-more-cache-less-cash/>

Memcached and MySQL caching are relatively advanced techniques that require some fine-tuning to really be of any use. So, what about the low-hanging fruit? Fortunately, some other techniques are quicker and easier to implement, starting with...

## Compressing Resources

Apache has a built-in way to selectively compress files on request<sup>184</sup>. In your `.htaccess` or, preferably, `httpd.conf` file, specify which content types should be compressed, like so:

```
<ifmodule mod_deflate.c>
    AddOutputFilterByType DEFLATE text/html text/plain
    text/xml text/css application/x-javascript
    application/javascript text/text
</ifmodule>
```

This specifies that the server should compress all files recognized as being HTML, XML, CSS, JavaScript and plain-text content types. Your server might not recognize other content types that should be compressed, such as OpenType, EOF and TrueType font files. If that is the case, you can specifically add those types using `mod_mime`:

```
<ifmodule mod_deflate.c>
    <ifmodule mod_mime.c>
        Addtype font/opentype .otf
        Addtype font/eot .eot
```

---

<sup>184</sup>. [http://httpd.apache.org/docs/2.2/mod/mod\\_deflate.html](http://httpd.apache.org/docs/2.2/mod/mod_deflate.html)

```
        Addtype font/truetype .ttf

</ifmodule>

AddOutputFilterByType DEFLATE text/html text/plain
text/xml text/css application/x-javascript
application/javascript text/text font/opentype
font/truetype font/eot

</ifmodule>
```

For additional compression steps and methods, check out [Viral Patel's adventures<sup>185</sup>](#) with Gzip, Deflate and PHP output buffering.

## Concatenate Files

Besides compressing resource files, you can also concatenate them in a variety of ways. Rob Flaherty documents a way to [concatenate all JavaScript files using PHP code<sup>186</sup>](#). Apache Ant enables you to configure a website build to [concatenate files<sup>187</sup>](#) without modifying the website's code. Note that this won't work if you deliver your resource files over the CDN! They are mutually exclusive techniques.

## Optimize JavaScript Loading

One of the simplest things you can do to accelerate your website's loading time is to move any `<script>` tags from

---

<sup>185</sup>. <http://viralpatel.net/blogs/>

[compress-php-css-js-javascript-optimize-website-performance/](http://viralpatel.net/blogs/compress-php-css-js-javascript-optimize-website-performance/)

<sup>186</sup>. <http://www.ravelrumba.com/blog/css-js-concatenation-versioning-php/>

<sup>187</sup>. <http://ant.apache.org/manual/Tasks/concat.html>

the `<head>` to the end of the HTML. As Google Developers explains<sup>188</sup>, any JavaScript that defines user interaction or modifies loaded HTML content (such as `onClick` and `onLoad` events) can be deferred until all other HTML has loaded. By moving JavaScript references out of the `<head>` tag, you cut down the time it takes for a user's browser to start loading the `<body>` and to actually display the content. The quicker you load the `<head>` tag and start displaying content, the better. Letting the user know that *something* is happening is preferable to showing a blank white screen for a second or two while all of those resource files load.

## *Assign Load Testing: DIY Or Outsourced?*

For large-scale national clients and enterprise applications, it's probably more efficient for a small development firm to outsource load testing to a specialist. The National Archives client outsourced this work to IBM, which acquired Rational Machines back in 2003 and is an industry leader in the full range of software testing — load, unit and accessibility testing, etc.

Take this opportunity to decide on your business' core competencies<sup>189</sup>. Ask yourself, how deep into the bytes will you venture? On the other hand, if your firm has decided to outsource load testing, please let us know in the comments section.

---

<sup>188</sup>. <https://developers.google.com/speed/docs/best-practices/payload#DeferLoadingJS>

<sup>189</sup>. <http://www.webperformance.com/load-testing/blog/2010/03/should-we-outsource-load-testing-or-do-it-ourselves/>

## *Bill Accordingly*

As your firm increases its core competencies to include load-time optimization, CDN content delivery and load testing, remember to account for the extra value that you add to your clients in the bottom line. Whether these services are line items in your standard contract or not, you'd be selling yourself short by failing to highlight them and citing them to justify your rates.

Slow page-loading times translate into frustrated users<sup>190</sup> and lost opportunities<sup>191</sup>: in sales, interaction, page visits, advertising impressions, etc. Most clients will glaze over if you go into too much technical detail, so keep your conversations focused on results. Discuss metrics with your client to figure out what they want out of the project: few clients will come to you asking to decrease their average page-loading time by 25%, but many will tell you they want to increase sales, receive more user feedback or attract eyeballs.

Your expertise is worth it. 🍷

---

<sup>190</sup>. <http://blog.kissmetrics.com/loading-time/>

<sup>191</sup>. <http://blog.tagman.com/2012/03/just-one-second-delay-in-page-load-can-cause-7-loss-in-customer-conversions/>



# Speed Up Your Mobile Website With Varnish

BY RACHEL ANDREW 🍷

Imagine that you have just written a post on your blog, tweeted about it and watched it get retweeted by some popular Twitter users, sending hundreds of people to your blog at once. Your excitement at seeing so many visitors talk about your post turns to dismay as they start to tweet that your website is down — a database connection error is shown.

Or perhaps you have been working hard to generate interest in your startup. One day, out of the blue, a celebrity tweets about how much they love your product. The person's followers all seem to click at once, and many of them find that the domain isn't responding, or when they try to sign up for the trial, the page times out. Despite your apologies on Twitter, many of the visitors move on with their day, and you lose much of the momentum of that initial tweet.

These scenarios are fairly common, and I have noticed in my own work that when content becomes popular via social networks, the proportion of mobile devices that access that content is higher than usual, because many people use their mobile devices, rather than desktop applications, to access Twitter and other social networks. Many of these mobile users access the Web via slow data connections and crowded public Wi-Fi. So, anything you can do to ensure that your website loads quickly will benefit those users.

In this chapter, I'll show you Varnish Web application accelerator<sup>192</sup>, a free and simple thing that makes a world of difference when a lot of people land on your website all at once.

## *Introducing The Magic*

For the majority of websites, even those whose content is updated daily, a large number of visitors are served exactly the same content. Images, CSS and JavaScript, which we expect not to change very much – but also content stored in a database using a blogging platform or content management system (CMS) – are often served to visitors in exactly the same way every time.

Visitors coming to a blog from Twitter would likely not all be served exactly the same content – including not only images, JavaScript and CSS, but also content that is created with PHP and with queries to the database before being served as a page to the browser. Each request for that blog's post would require not only the Web server that serves the file (for example, Apache), but also PHP scripts, a connection to the database, and queries run against database tables.

The number of database connections that can be made and the number of Apache processes that can run are always limited. The greater the number of visitors, the less memory available and the slower each request becomes. Ultimately, users will start to see database connection errors, or the website will just seem to hang, with pages not

---

<sup>192</sup>. <https://www.varnish-cache.org/>

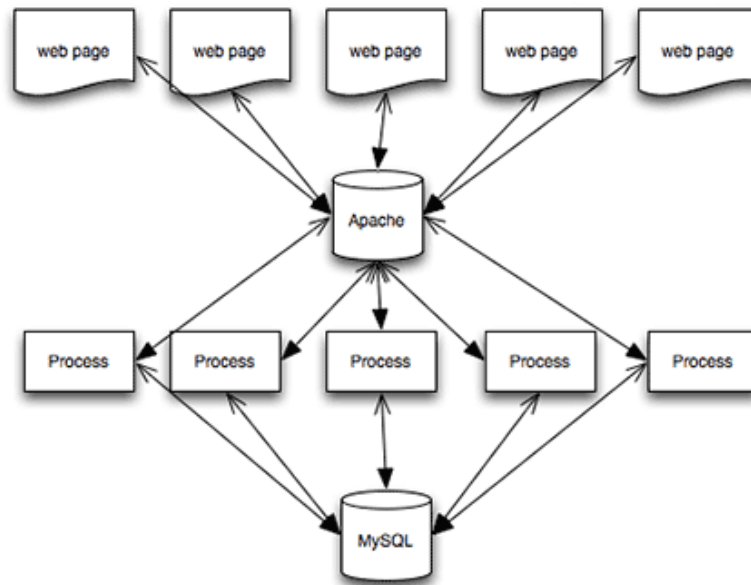
loading as the server struggles to keep up with demand.

This is where an HTTP cache like Varnish comes in. Instead of requests from browsers directly hitting your Web server, making the server create and serve the pages requested, requests would first hit the cache. If the requested page is in the cache, then it is served directly from memory, never touching Apache or the database. If the page is not in the cache, then the request is handed over to Apache as usual, whereupon Apache will create and serve the page, which is then stored in the cache, ready for the next request.

Serving a page from memory is a lot faster than serving it from disk via Apache. In addition, the page never needs to touch PHP or the database, leaving those processes free to handle traffic that does require a database connection or some processing. For example, in our second scenario of a startup being mentioned by a celebrity, the majority of people clicking through would check out only a few pages of the website — all of those pages could be in the cache and served from memory. The few who go on to sign up would find that the registration form works well, because the server-side code and database connection are not bogged down by people pouring in from Twitter.

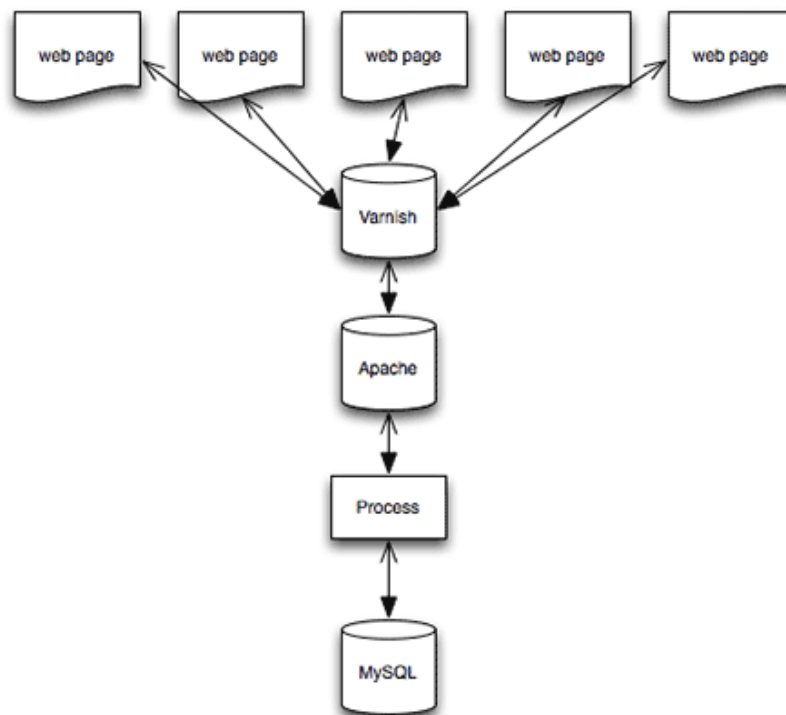
## *How Does It Work?*

The diagram below shows how a blog post might be served when all requests go to the Apache Web server. This example shows five browsers all requesting the same page, which uses PHP and MySQL.



Every HTTP request is served by Apache — images, CSS, JavaScript and HTML files. If a file is PHP, then it is parsed by PHP. And if content is required from the database, then a database connection is made, SQL queries are run, and the page is assembled from the returned data before being served to the browser via Apache.

If we place Varnish in front of Apache, we would instead see the following:



If the page and assets requested are already cached, then Varnish serves them from memory – Apache, PHP and MySQL would never be touched. If a browser requests something that is not cached, then Varnish hands it over to Apache so that it can do the job detailed above. The key point is that Apache needs to do that job only once, because the result is then stored in memory, and when a second request is made, Varnish can serve it.

The tool has other benefits. In Varnish terminology, when you configure Apache as your Web server, you are configuring a “back end.” Varnish allows you to configure multiple back ends. So, you might want to run two Web servers – for example, using Apache for PHP pages while serving static assets (such as CSS files) from nginx. You can set this up in Varnish, which will pass the request

through to the correct server. In this tutorial, we will look at the simplest use case.

## *I'm Sold! How Do I Get Started?*

Varnish is really easy to install and configure. You will need root, or **sudo**, access to your server to install things on it. Therefore, your website needs to be hosted on a virtual private server (VPS) or the like. You can get a VPS very inexpensively these days, and Varnish is a big reason to choose a VPS over shared hosting.

Some CMS' have plugins that work with Varnish or that integrate it in the control panel – usually to make clearing the cache easier. But you can put Varnish in any CMS or any static website, without any particular integration with other systems.

I'll walk you through installing Varnish, assuming that you already run Apache as a Web server on your system. I run Debian Linux, but packages for other distributions are available. (The paths to files on the system will vary with the Linux distribution.)

Before starting, check that Apache is serving your website as expected. If the server is brand new or you are trying out Varnish on a local virtual machine, make sure to configure a virtual host and that you can view a test page on the server using a browser.

## *Install Varnish*

Installation instructions for various platforms are in [Varnish's documentation](#)<sup>193</sup>. I am using Debian Wheezy; so,

as root, I followed the [instructions for Debian](#)<sup>194</sup>. Once Varnish is installed, you will see the following line in the terminal, telling you that it has started successfully.

```
[ ok ] Starting HTTP accelerator: varnishd.
```

By default, Apache listens for requests on port **80**. This is where incoming HTTP requests go, because we want Varnish to essentially sit in front of Apache. We need to configure Varnish to listen on port **80** and change Apache to a different port – usually **8080**. We then tell Varnish where Apache is.

## *Reconfigure Apache*

To change the port that Apache listens on, open the file `/etc/apache2/ports.conf` as root, and find the following lines:

```
NameVirtualHost *:80
Listen 80
```

Change these lines to this:

```
NameVirtualHost *:8080
Listen 8080
```

If you see the following lines, just change **80** to **8080** in the same way.

---

<sup>193</sup>. <https://www.varnish-cache.org/docs>

<sup>194</sup>. <https://www.varnish-cache.org/installation/debian>

```
NameVirtualHost 127.0.0.1:80
Listen 80
```

Save this file and open your default virtual host file, which should be in `/etc/apache2/sites-available`. In this file, find the following line:

```
<VirtualHost *:80>
```

Change it to this:

```
<VirtualHost *:8080>
```

You will also need to make this change to any other virtual hosts you have set up.

## Configure Varnish

Open the file `/etc/default/varnish`, and scroll down to the uncommented section that starts with `DAEMON_OPTS`. Edit this so that it looks like the following block, which will make Varnish listen on port `80`.

```
DAEMON_OPTS="-a :80
-T localhost:1234
-f /etc/varnish/default.vcl
-S /etc/varnish/secret
-s malloc,256m"
```

Open the file `/etc/varnish/default.vcl`, and check that the default back end is set to port `8080`, because this is where Apache will be now.



```
backend default {  
    .host = "127.0.0.1";  
    .port = "8080";  
}
```

Restart Apache and Varnish as root with the following commands:

```
service apache2 restart  
service varnish restart
```

Check that your test website is still available. If it is, then you'll probably be wondering how to test that it is being served from Varnish. There are a few ways to do this. The simplest is to use cURL. In the command line, type the following:

```
curl http://yoursite.com --head
```

The response should be something like **Via: 1.1 varnish**.

You can also look at the statistics generated by Varnish. In the command line, type **varnishstat**, and watch the hit rate increase as you refresh your page in the browser. Varnish refers to something it can serve as a “hit” and something it passes to Apache or another backend as a “miss.”

Another useful tool is varnish-top. Type **varnishtop -i txurl** in the command line, and refresh your page in the browser. This tool shows you which files are being served by Varnish.

## *Purging The Cache*

Now that pages are being cached, if you change an HTML or CSS file, you won't see the changes immediately. This trips me up all of the time. I know that a cache is in front of Apache, yet every so often I still have that baffled moment of "Where are my changes?!" Type `varnishadm "ban.url ."` in the command line to clear the entire cache.

You can also control Varnish over HTTP. Plugins are available, such as Varnish HTTP Purge<sup>195</sup> for WordPress, that you can configure to purge the cache directly from the administration area.

## *Some Simple Customizations*

You'll probably want to know a few things about how Varnish works by default in order to tweak it. Configuring it as described above should cause most basic assets and pages to be served from the cache, once those assets have been cached in memory.

Varnish will only cache things that are safe to do so, and it might not cache some common things that you think it would. A good example is cookies.

In its default configuration, Varnish will not cache content if a cookie is set. So, if your website serves different content to logged-in users, such as personalized content, you wouldn't want to serve everyone content that is meant for one user. However, you'd probably want to ig-

---

<sup>195</sup>. <http://wordpress.org/plugins/varnish-http-purge/>

nore some cookies, such as for analytics. If the website does not serve any personalized content, then the only cookies you would probably care about are those set for your admin area — it would be inconvenient if Varnish cached the admin area and you couldn't see changes.

Let's edit `/etc/varnish/default.vcl`. Assuming your admin area is at `/admin`, you would add the following:

```
sub vcl_recv {  
    if ( !( req.url ~ ^/admin/ ) ) {  
        unset req.http.Cookie;  
    }  
}
```

Some cookies might be important — for example, logged-in users should get uncached content. So, you don't want to eliminate all cookies. A trip to the land of regular expressions is required to identify the cookies we'll need. Many recipes for doing this can be found with a quick search online. For analytics cookies, you could add the following.

```
sub vcl_recv {  
    // Remove has_js and Google Analytics __* cookies.  
    set req.http.Cookie = regsuball(req.http.Cookie,  
    "(^|;s*)(_[a-z]+|has_js)=[^;]*", "");  
    // Remove a ";" prefix, if present.  
    set req.http.Cookie = regsub(req.http.Cookie, "^;s*", "");  
}
```

Varnish has a section in its documentation on “Cookies<sup>196</sup>.”

In most cases, configuring Varnish as described above and removing analytics cookies will dramatically speed up your website. Once Varnish is up and running and you are familiar with the logs, you can start to tweak the configuration and get more performance from the cache.

## Next Steps

To learn more, go through [Varnish's documentation](#)<sup>197</sup>. You should understand enough of Varnish's basics by now to try some of the examples. The section on "[Achieving a High Hit Rate](#)"<sup>198</sup> is well worth a read for the simple tips on tweaking your configuration. 🐣



*Keep calm and try Varnish to optimize mobile websites.  
(Image: [Varnish Cache](#)<sup>199</sup>)*

---

<sup>196</sup>. <https://www.varnish-cache.org/docs/3.0/tutorial/cookies.html>

<sup>197</sup>. <https://www.varnish-cache.org/docs/3.0/tutorial/index.html>

<sup>198</sup>. [https://www.varnish-cache.org/docs/3.0/tutorial/increasing\\_your\\_hitrate.html](https://www.varnish-cache.org/docs/3.0/tutorial/increasing_your_hitrate.html)

<sup>199</sup>. <https://twitter.com/varnishcache>

# Cache Invalidation

## Strategies With Varnish

### Cache

BY PER BUER 🍷

Phil Karlton once said, “There are only two hard things in Computer Science: cache invalidation and naming things.” This chapter is about the harder of these two: cache invalidation. It’s directed at readers who already work with Varnish Cache. To learn more about it, you’ll find background information in “Speed Up Your Mobile Website With Varnish.”

*10 microseconds (or 250 milliseconds):* That’s the difference between delivering a cache hit and delivering a cache miss. How often you get the latter will depend on the efficiency of the cache — this is known as the “hit rate.” A cache miss depends on two factors: the volume of traffic and the average time to live (TTL), which is a number indicating how long the cache is allowed to keep an object. As system administrators and developers, we can’t do much about the traffic, but we can influence the TTL.

However, to have a high TTL, we need to be able to invalidate objects from the cache so that we avoid serving stale content. With Varnish Cache, there are myriad ways to do this. We’ll explore the most common ways and how to deploy them.

Varnish does a whole lot of other stuff as well, but its caching services are most popular. Caches speed up Web services by serving cached static content. When Varnish

Cache is delivering a cache hit, it usually just dumps a chunk of memory into a socket. Varnish Cache is so fast that, on modern hardware, we actually measure response time in microseconds!



*Caching isn't always as simple as we think; a few gotchas and problems may take quite some of our time to master. (Image: [Varnish Cache](https://twitter.com/varnishcache)<sup>200</sup>)*

When using a cache, you need to know when to evict content from the cache. If you have no way to evict content, then you would rely on the cache to time-out the object after a predetermined amount of time. This is one method, but hardly the most optimal solution. The best way would be to let Varnish Cache keep the object in memory forever (mostly) and then tell the object when to refresh. Let's go into detail on how to achieve this.

---

<sup>200</sup>. <https://twitter.com/varnishcache>

## HTTP Purging

HTTP Purging is the most straightforward of these methods. Instead of sending a **GET** */url* to Varnish, you would send **PURGE** */url*. Varnish would then discard that object from the cache. Add an access control list to Varnish so that not just anyone can purge objects from your cache; other than that, though, you're home free.

```
acl purge {  
    "localhost";  
    "192.168.55.0"/24;  
}  
  
sub vcl_recv {  
    # allow PURGE from localhost and 192.168.55...  
  
    if (req.request == "PURGE") {  
        if (!client.ip ~ purge) {  
            error 405 "Not allowed.";  
        }  
        return (lookup);  
    }  
}  
  
sub vcl_hit {  
    if (req.request == "PURGE") {  
        purge;  
        error 200 "Purged.";  
    }  
}
```

```

sub vcl_miss {
    if (req.request == "PURGE") {
        purge;
        error 200 "Purged.";
    }
}

```

## SHORTCOMINGS OF PURGING

HTTP purging falls short when a piece of content has a complex relationship to the URLs it appears on. A news article, for instance, might show up on a number of URLs. The article might have a desktop view and a mobile view, and it might show up on a section page and on the front page. Therefore, you would have to either get the content management system to keep track of all of these manifestations or let Varnish do it for you. To let Varnish do it, you would use bans, which we'll get into now.

## *Bans*

A ban is a feature specific to Varnish and one that is frequently misunderstood. It enables you to ban Varnish from serving certain content in memory, forcing Varnish to fetch new versions of these pages.

An interesting aspect is how you specify which pages to ban. Varnish has a language that provides quite a bit of flexibility. You could tell Varnish to ban by giving the ban command in the command-line interface, typically connecting to it with **varnishadm**. You could also do it



through the Varnish configuration language (VCL), which provides a simple way to implement HTTP-based banning.

Let's start with an example. Suppose we need to purge our website of all images.

```
> ban obj.http.content-type ~ "^image/"
```

The result of this is that, for all objects in memory, the HTTP response header **Content-Type** would match the regular expression **^image/**, which would invalidate immediately.

Here's what happens in Varnish. First, the ban command puts the ban on the "ban list." When this command is on the ban list, every cache hit that serves an object older than the ban itself will start to look at the ban list and compare the object to the bans on the list. If the object matches, then Varnish kills it and fetches a newer one. If the object doesn't match, then Varnish will make a note of it so that it does not check again.

Let's build on our example. Now, we'll only ban images that are placed somewhere in the **/feature** URL. Note the logical "and" operator, **&&**.

```
> ban obj.http.content-type ~ "^image/" && req.url ~  
"^\feature"
```

You'll notice that it says **obj.http.content-type** and **req.url**. In the first part of the ban, we refer to an attribute of an object stored in Varnish. In the latter, we refer to a part of a request for an object. This might be a bit unconventional, but you can actually use attributes on the

request to invalidate objects in cache. Now, `req.url` isn't normally stored in the object, so referring to the request is the only thing we can do here. You could use this to do crazy things, like ban everything being requested by a particular client's IP address, or ban everything being requested by the Chromium browser. As these requests hit Varnish, objects are invalidated and refreshed from the originating server.

Issuing bans that depend on the request opens up some interesting possibilities. However, there is one downside to the process: A very long list of bans could slow down content delivery.

There is a worker thread assigned to the task of shortening the list of bans, "the ban lurker". The ban lurker tries to match a ban against applicable objects. When a ban has been matched against all objects older than itself, it is discarded.

As the ban lurker iterates through the bans, it doesn't have an HTTP request that it is trying to serve. So, any bans that rely on data from the request cannot be tested by the ban lurker. To keep ban performance up, then, we would recommend not using request data in the bans. If you need to ban something that is typically in the request, like the URL, you can copy the data from the request to the object in `vcl_fetch`, like this:

```
set beresp.http.x-url = req.url;
```

Now, you'll be able to use bans on `obj.http.x-url`. Remember that the `beresp` objects turn into `obj` as it gets stored in cache.

## Tagging Content For Bans

Bans are often a lot more effective when you give Varnish a bit of help. If the object has an **X-Article-id** header, then you don't need to know all of the URLs that the object is presented as.

For pages that depend on several objects, you could have the content management system add an **X-depends-on** header. Here, you'd list the objects that should trigger an update of the current document. To take our news website again, you might use this to list all articles mentioned on the front page:

```
X-depends-on: 3483 4376 32095 28372
```

Naturally, then, if one of the articles changes, you would issue a ban, like this:

```
ban obj.http.x-depends-on ~ "\D4376\D"
```

This is potentially very powerful. Imagine making the database issue these invalidation requests through triggers, thus eliminating the need to change the middleware layer. Neat, eh?

## Graceful Cache Invalidation

Imagine purging something from Varnish and then the origin server that was supposed to replace the content suddenly crashes. You've just thrown away your only workable copy of the content. What have you done?! Turns out that quite a few content management systems crash on a regular basis.

Ideally, we would want to put the object in a third state — to invalidate it on the condition that we’re able to get some new content. This third state exists in Varnish: It is called “grace,” and it is used with TTL-based invalidations. After an object expires, it is kept in memory in case the back-end server crashes. If Varnish can’t talk to the back end, then it checks to see whether any graced objects match, and it serves those instead.

One Varnish module (or VMOD), named **softpurge**, allows you to invalidate an object by putting it into the grace state. Using it is simple. Just replace the **PURGE** VCL with the VCL that uses the **softpurge** VMOD.

```
import softpurge;

sub vcl_hit {
    if (req.method == "PURGE") {
        softpurge.softpurge();
        error 200 "Successful softpurge";
    }
}

sub vcl_miss {
    if (req.method == "PURGE") {
        softpurge.softpurge();
        error 200 "Successful softpurge";
    }
}
```

## Distributing Cache Invalidations Events

All of the methods listed above describe the process of invalidating content on a single cache server. Most serious configurations would have more than one Varnish server. If you have two, which should give enough oomph for most websites, then you would want to issue one invalidation event for each server. However, if you have 20 or 30 Varnish servers, then you really wouldn't want to bog down the application by having it loop through a huge list of servers.

Instead, you would want a single API end point to which you can send your purges, having it distribute the invalidation event to all of your Varnish servers. For reference, here is a very simple invalidation service written in shell script. It will listen on port 2000 and invalidate URLs to three different servers (*alfa*, *beta* and *gamma*) using cURL.

```
nc -l 2000 | while true
do read url
for srv in "alfa" "beta" "gamma"
do curl -m 2 -x $srv -X PURGE $url
done
done
```

It might not be suitable for production because the error handling leaves something to be desired!

Cache invalidation is almost as important as caching. Therefore, having a sound strategy for invalidating the content is crucial to maintaining high performance and having a high cache-hit ratio. If you maintain a high hit

rate, then you'll need fewer servers and will have happier users and probably less downtime. With this, you're hopefully more comfortable using tools like these to get stale content out of your cache. 🍷

# Gone In 60 Frames Per Second: A Pinterest Paint Performance Case Study

BY ADDY OSMANI 🍷

Today we'll discuss how to improve the paint performance of your websites and Web apps. This is an area that we Web developers have only recently started looking at more closely, and it's important because it could have an impact on your user engagement and user experience.

## *Frame Rate Applies To The Web, Too*

Frame rate is the rate at which a device produces consecutive images to the screen. A low frames per second (FPS) means that individual frames can be made out by the eye. A high FPS gives users a more responsive feel. You're probably used to this concept from the world of gaming, but it applies to the Web, too.

Long image decoding, unnecessary image resizing, heavy animation and data processing can all lead to dropped frames, which reduces the frame rate, resulting in janky pages. We'll explain what exactly we mean by "jank" shortly.

## *Why Care About Frame Rate?*

Smooth, high frame rates drive user engagement and can affect how much users interact with your website or app.

At EdgeConf earlier this year, Facebook confirmed<sup>201</sup> this when it mentioned that in an A/B test, it slowed down scrolling from 60 FPS to 30 FPS, causing engagement to collapse. That said, if you can't do high frame rates and 60 FPS is out of reach, then you'd at least want something smooth. If you're doing your own animation, this is one benefit of using `requestAnimationFrame`<sup>202</sup>: the browser can dynamically adjust to keep the frame rate normal.

In cases where you're concerned about scrolling, the browser can manage the frame rate for you. But if you introduce a large amount of jank, then it won't be able to do as good a job. So, try to avoid big hitches, such as long paints, long JavaScript execution times, long anything.

## *Don't Guess It, Test It!*

Before getting started, we need to step back and look at our approach. We all want our websites and apps to run more quickly. In fact, we're arguably paid to write code that runs not only correctly, but quickly. As busy developers with deadlines, we find it very easy to rely on snippets of advice that we've read or heard. Problems arise when

---

<sup>201</sup>. [http://www.youtube.com/watch?list=SPNYkxOF6rcICCU\\_UD67GaoqLvMjnBBwft&v=3-WYu\\_p5rdU&feature=player\\_detailpage#t=2149s](http://www.youtube.com/watch?list=SPNYkxOF6rcICCU_UD67GaoqLvMjnBBwft&v=3-WYu_p5rdU&feature=player_detailpage#t=2149s)

<sup>202</sup>. <https://developer.mozilla.org/en-US/docs/Web/API/window.requestAnimationFrame>



we do that, though, because the internals of browsers change very rapidly, and something that's slow today could be quick tomorrow.

Another point to remember is that your app or website is unique, and, therefore, the performance issues you face will depend heavily on what you're building. Optimizing a game is a very different beast to optimizing an app that users will have open for 200+ hours. If it's a game, then you'll likely need to focus your attention on the main loop and heavily optimize the chunk of code that is going to run every frame. With a DOM-heavy application, the memory usage might be the biggest performance bottleneck.

Your best option is to learn how to measure your application and understand what the code is doing. That way, when browsers change, you will still be clear about what matters to you and your team and will be able to make informed decisions. So, no matter what, don't guess it, test it!<sup>203</sup>

We're going to discuss how to measure frame rate and paint performance shortly, so hold onto your seats!

**Note:** Some of the tools mentioned in this chapter require Chrome Canary<sup>204</sup>, with the “Developer Tools experiments” enabled in **about:flags**. (We — Addy Osmani and Paul Lewis — are engineers on the Developer Relations team at Chrome.)

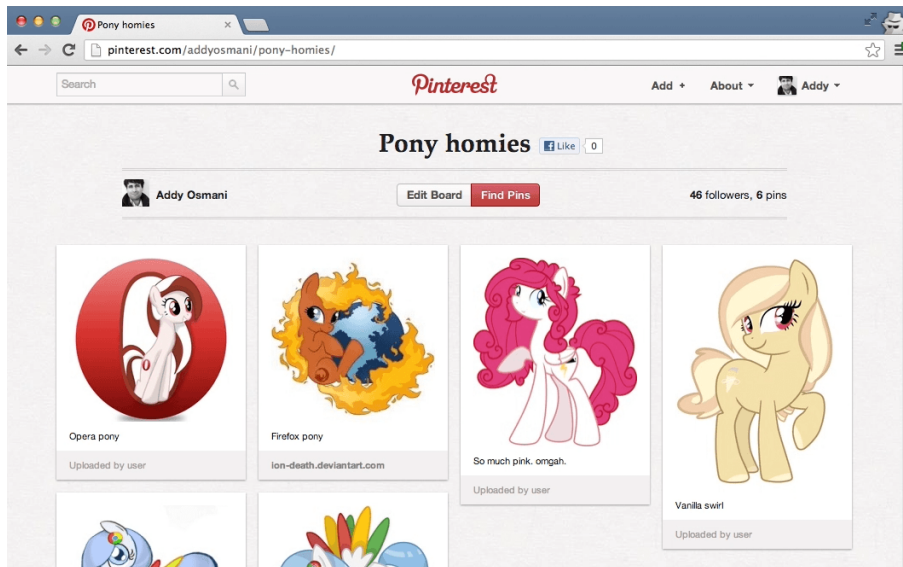
---

<sup>203</sup>. <http://aerotwist.com/blog/dont-guess-it-test-it/>

<sup>204</sup>. <https://www.google.com/intl/en/chrome/browser/canary.html>

## Case Study: Pinterest

The other day we were on [Pinterest](http://pinterest.com/)<sup>205</sup>, trying to find some ponies to add to our pony board (Addy loves ponies!). So, we went over to the Pinterest feed and started scrolling through, looking for some ponies to add.



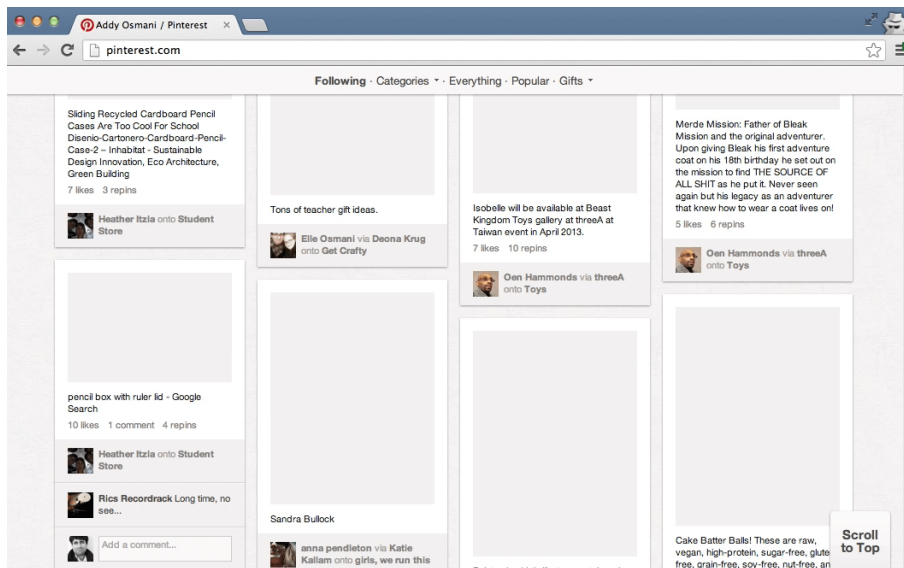
*Addy adding some ponies to his Pinterest board, as one does.*

### JANK AFFECTS USER EXPERIENCE

The first thing we noticed as we scrolled was that scrolling on this page doesn't perform very well — scrolling up and down takes effort, and the experience just feels sluggish. When they come up against this, users get frustrated, which means they're more likely to leave. Of course, this is the last thing we want them to do!

---

<sup>205</sup>. <http://pinterest.com/>



*Pinterest showing a performance bottleneck when a user scrolls.  
(Animated GIF<sup>206</sup>)*

This break in consistent frame rate is something the Chrome team calls “jank,” and we’re not sure what’s causing it here. You can actually notice some of the frames being drawn as we scroll. But let’s visualize it! We’re going to open up Frames mode and show what slow looks like there in just a moment.

**Note:** What we’re really looking for is a consistently high FPS, ideally matching the refresh rate of the screen. In many cases, this will be 60 FPS, but it’s not guaranteed, so check the devices you’re targeting.

Now, as JavaScript developers, our first instinct is to suspect a memory leak as being the cause. Perhaps some ob-

---

<sup>206</sup>. [http://media.smashingmagazine.com/wp-content/uploads/2013/05/slow\\_scroll2.gif](http://media.smashingmagazine.com/wp-content/uploads/2013/05/slow_scroll2.gif)

jects are being held around after a round of garbage collection. The reality, however, is that very often these days JavaScript is not a bottleneck. Our major performance problems come down to slow painting and rendering times. The DOM needs to be turned into pixels on the screen, and a lot of paint work when the user scrolls could result in a lot of slowing down.

**Note:** HTML5 Rocks specifically discusses some of the causes of slow scrolling<sup>207</sup>. If you think you’re running into this problem, it’s worth a read.

## *Measuring Paint Performance*

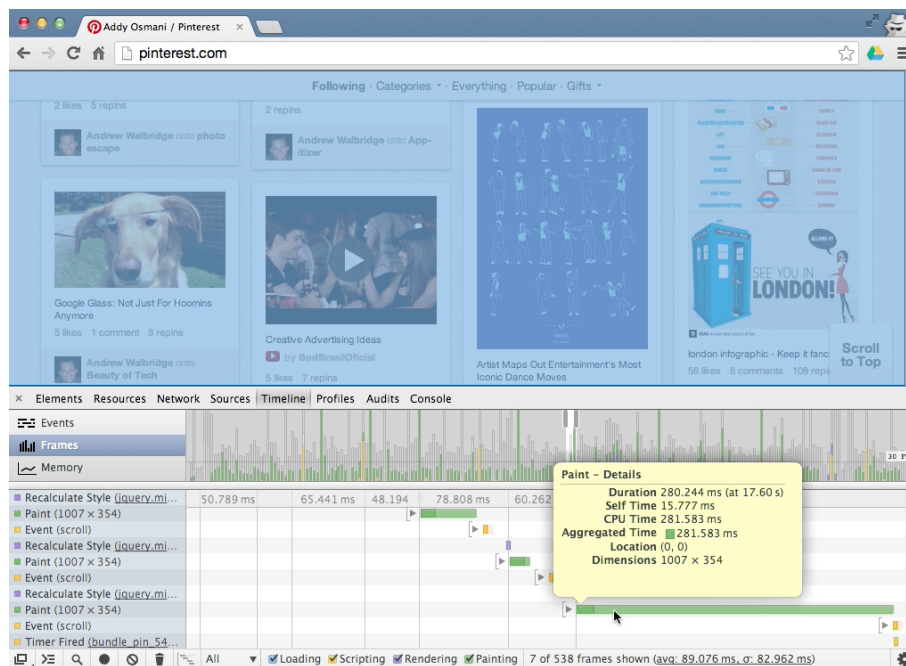
### **FRAME RATE**

We suspect that something on this page is affecting the frame rate. So, let’s go open up Chrome’s Developer Tools and head to the “Timeline” and “Frames” mode to record a new session. We’ll click the record button and start scrolling the page the way a normal user would. Now, to simulate a few minutes of usage, we’re going to scroll just a little faster.

Up, down, up, down. What you’ll notice now in the summary view up at the top is a lot of purple and green, corresponding to painting and rendering times. Let’s stop recording for now. As we flip through these various frames, we see some pretty hefty “Recalculate Styles” and a lot of “Layout.”

---

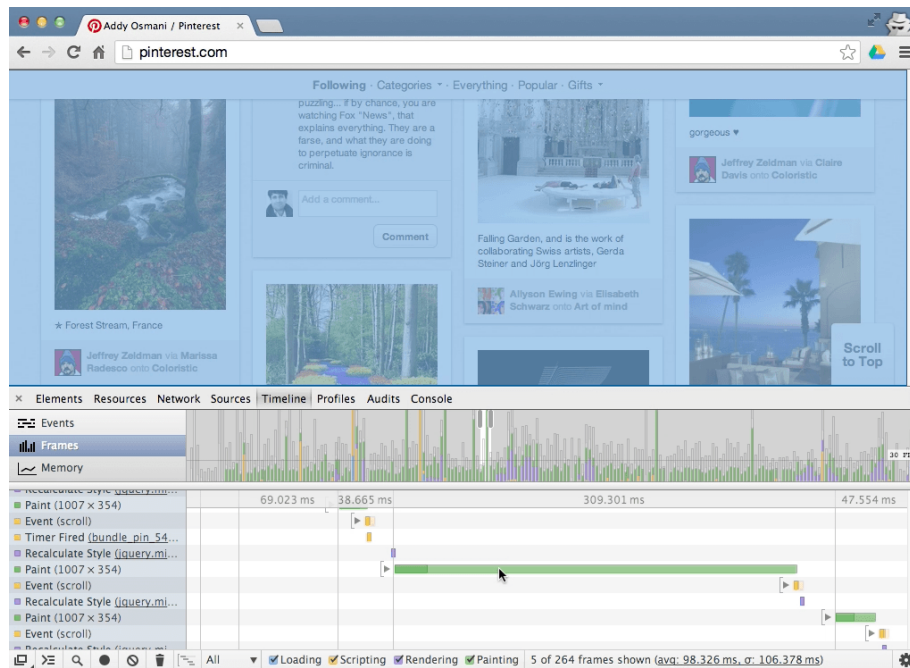
<sup>207</sup>. <http://www.html5rocks.com/en/tutorials/speed/scrolling/>



Using Chrome's Developer Tools to profile scrolling interactions.  
(Animated GIF<sup>208</sup>)

If you look at the legend to the very right, you'll see that we've actually blown our budget of 60 FPS, and we're not even hitting 30 FPS either in many cases. It's just performing quite poorly. Now, each of these bars in the summary view correspond to one frame — i.e. all of the work that Chrome has to do in order to be able to draw an app to the screen.

<sup>208</sup>. <http://media.smashingmagazine.com/wp-content/uploads/2013/05/performance.gif>



Chrome's Developer Tools showing a long paint time. (Animated GIF<sup>209</sup>)

## FRAME BUDGET

If you're targeting 60 FPS, which is generally the optimal number of frames to target these days, then to match the refresh rate of the devices we commonly use, you'll have a 16.7-millisecond budget in which to complete everything — JavaScript, layout, image decoding and resizing, painting, compositing — everything.

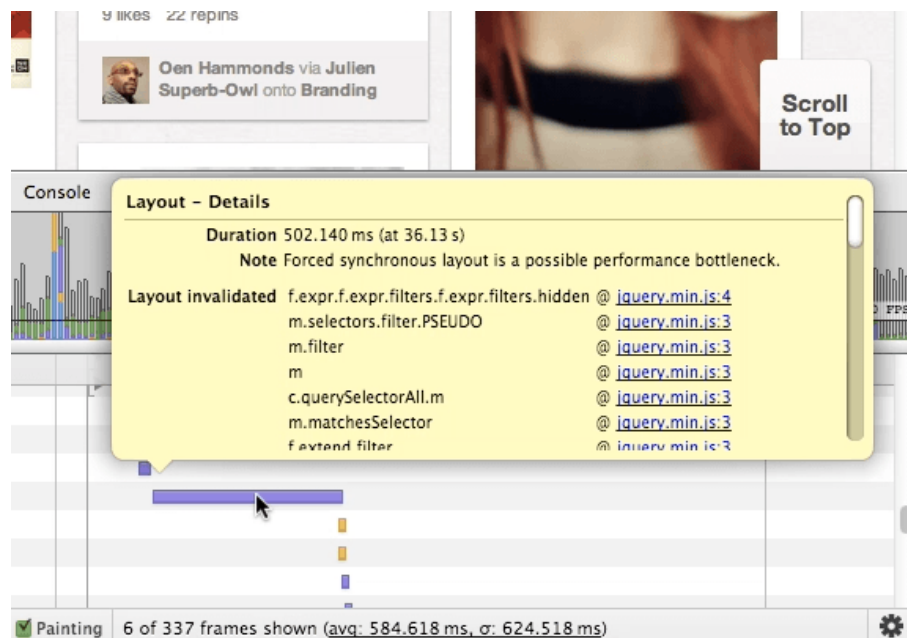
**Note:** A constant frame rate is our ideal here. If you can't hit 60 FPS for whatever reason, then you're likely better off targeting 30 FPS, rather than allowing a variable frame rate between 30 and 60 FPS. In practice, this can be challenging to code because when the JavaScript finishes

---

209. <http://media.smashingmagazine.com/wp-content/uploads/2013/05/selection1.gif>

executing, all of the layout, paint and compositing work still has to be done, and predicting that ahead of time is very difficult. In any case, whatever your frame rate, ensure that it is consistent and doesn't fluctuate (which would appear as stuttering).

If you're aiming for low-end devices, such as mobile phones, then that frame budget of 16 milliseconds is really more like 8 to 10 milliseconds. This could be true on desktop as well, where your frame budget might be lowered as a result of miscellaneous browser processes. If you blow this budget, you will miss frames and see jank on the page. So, you likely have somewhere nearer 8 to 10 milliseconds, but be sure to test the devices you're supporting to get a realistic idea of your budget.



*An extremely costly layout operation of over 500 milliseconds.  
(Animated GIF<sup>210</sup>)*

**Note:** We’ve also got an article on how to use the Chrome Developer Tools to find and fix rendering performance issues<sup>211</sup> that focuses more on the timeline.

Going back to scrolling, we have a sneaking suspicion that a number of unnecessary repaints are occurring on this page with `onscroll`.

One common mistake is to stuff just way too much JavaScript into the `onscroll` handlers of a page — making it difficult to meet the frame budget at all. Aligning the work to the rendering pipeline (for example, by placing it in `requestAnimationFrame`) gives you a little more headroom, but you still have only those few milliseconds in which to get everything done.

The best thing you can do is just capture values such as `scrollTop` in your scroll handlers, and then use the most recent value inside a `requestAnimationFrame` callback.

## PAINT RECTANGLES

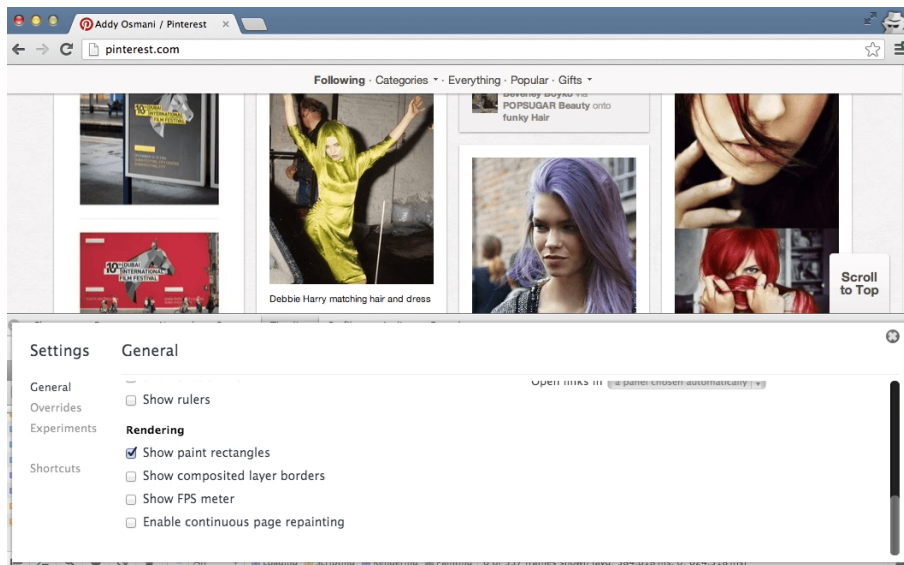
Let’s go back to `Developer Tools` → `Settings` and enable “Show paint rectangles.” This visualizes the areas of the screen that are being painted with a nice red highlight. Now look at what happens as we scroll through Pinterest.

---

<sup>210</sup>. <http://media.smashingmagazine.com/wp-content/uploads/2013/05/highlight.gif>

<sup>211</sup>. <http://addyosmani.com/blog/performance-optimisation-with-timeline-profiles/>

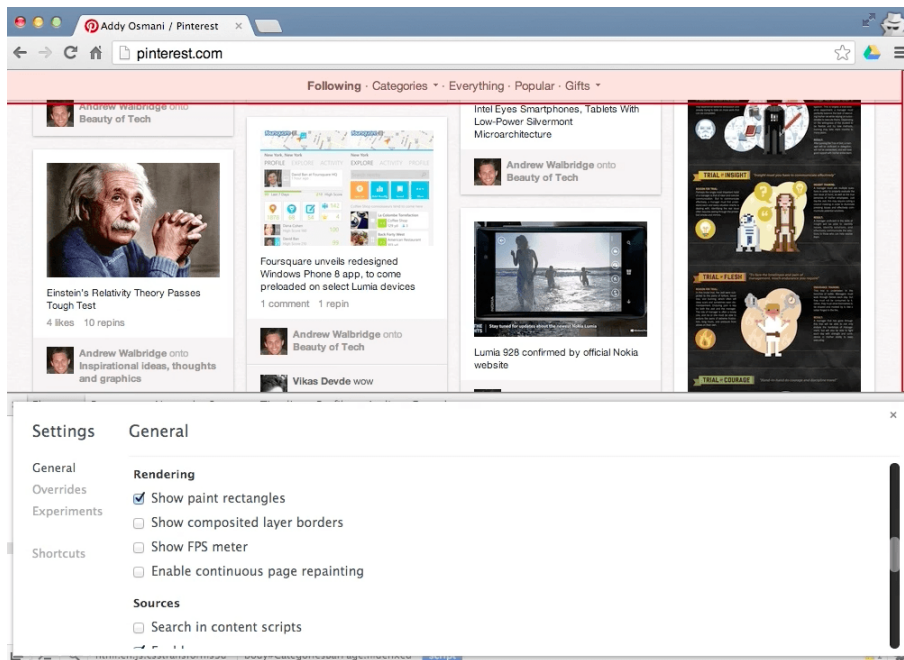




*Enabling Chrome Developer Tools' "Paint Rectangles" feature.  
(Animated GIF<sup>212</sup>)*

Every few milliseconds, we experience a big bright flash of red across the entire screen. There seems to be a paint of the whole screen every time we scroll, which is potentially very expensive. What we want to see is the browser just painting what is new to the page — so, typically just the bottom or top of the page as it gets scrolled into view. The cause of this issue seems to be the little “scroll to top” button in the lower-right corner. As the user scrolls, the fixed header at the top needs to be repainted, but so does the button. The way that Chrome deals with this is to create a union of the two areas that need to be repainted.

<sup>212</sup>. <http://media.smashingmagazine.com/wp-content/uploads/2013/05/rects.gif>

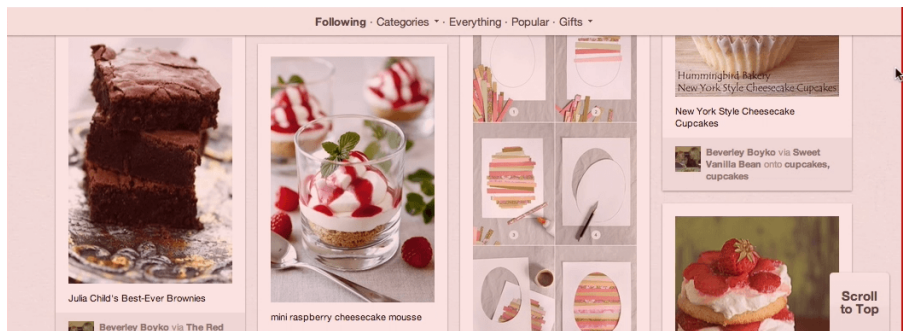


*Chrome shows freshly painted areas with a red box.*

In this case, there is a rectangle from the top left to top right, but not very tall, plus a rectangle in the lower-right corner. This leaves us with a rectangle from the top left to bottom right, which is essentially the whole screen! If you inspect the button element in Developer Tools and either hide it (using the **H** key) or delete it and then scroll again, you will see that only the header area is repainted. The way to solve this particular problem is to move the scroll button to its own layer so that it doesn't get unioned with the header. This essentially isolates the button so that it can be composited on top of the rest of the page. But we'll talk about layers and compositing in more detail in a little bit.

The next thing we notice has to do with hovering. When we hover over a pin, Pinterest paints an action bar containing “Repin, comment and like” buttons — let's call

this the action bar. When we hover over a single pin, it paints not just the bar but also the elements underlying it. Painting should happen only on those elements that you expect to change visually.



*A cause for concern: full-screen flashes of red indicate a lot of painting.  
(Animated GIF<sup>213</sup>)*

There's another interesting thing about scrolling here. Let's keep our cursor hovered over this pin and start scrolling the page again.

Every time we scroll through a new row of images, this action bar gets painted on yet another pin, even though we don't mean to hover over it. This comes down more to UX than anything else, but scrolling performance in this case might be more important than the hover effect during scrolling. Hovering amplifies jank during scrolling because the browser essentially pauses to go off and paint the effect (the same is true when we roll out of the element!). One option here is to use a `setTimeout` with a delay to ensure that the bar is painted only when the user really intends to use it, an approach we covered in

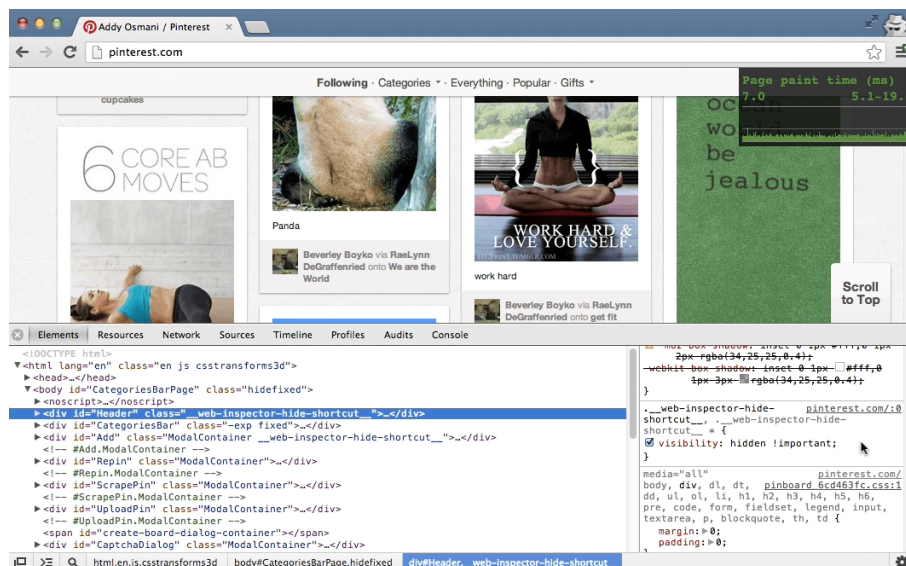
---

<sup>213</sup> <http://media.smashingmagazine.com/wp-content/uploads/2013/05/scroll.gif>

“Avoiding Unnecessary Paints<sup>214</sup>.” A more aggressive approach would be to measure the **mouseenter** or the mouse’s trajectory before enabling hover behaviors. While this measure might seem rather extreme, remember that we are trying to avoid unnecessary paints at all costs, especially when the user is scrolling.

## OVERALL PAINT COST

We now have a really great workflow for looking at the overall cost of painting on a page; go back into Developer Tools and “Enable continuous page repainting.”



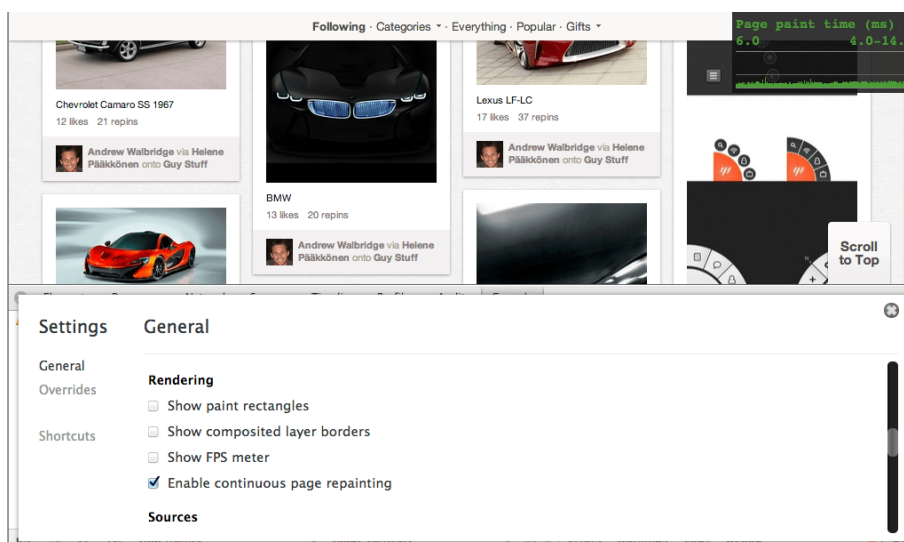
Chrome’s “Continuous Page Repainting” mode helps you to assess the overall cost of a page. (Animated GIF<sup>215</sup>)

<sup>214</sup>. <http://www.html5rocks.com/en/tutorials/speed/unnecessary-paints/>

<sup>215</sup>. <http://media.smashingmagazine.com/wp-content/uploads/2013/05/painthud.gif>

This feature will constantly paint to your screen so that you can find out what elements have costly paint times. You'll get this really nice black box in the top corner that summarizes paint times, with the minimum and maximum also displayed.

Let's head back to the "Elements" panel. Here, we can select a node and just use the keyboard to walk the DOM tree. If we suspect that an element has an expensive paint, we can use the **H** shortcut key (something recently added to Chrome) to toggle visibility on that element. Using the continuous paint box, we can instantly see whether this has a positive effect on our pages' paint times. We should expect it to in many cases, because if we hide an element, we should expect a corresponding reduction in paint times. But by doing this, we might see one element that is especially expensive, which would bear further scrutiny!



The "Continuous Page Repainting" chart showing the time taken to paint the page. (Animated GIF<sup>216</sup>)

For Pinterest’s website, we can do it to the categories bar or to the header, and, as you’d expect, because we don’t have to paint these elements at all, we see a drop in the time it takes to paint to the screen. If we want even more detailed insight, we can go right back to the timeline and record a new session to measure the impact. Isn’t that great? Now, while this workflow should work great for most pages, there might be times when it isn’t as useful. In Pinterest’s case, the pins are actually quite deeply nested in the page, making it hard for us to measure paint times in this workflow.

Luckily, we can still get some good mileage by selecting an element (such as a pin here), going to the “Styles” panel and looking at what CSS styles are being used. We can toggle properties on and off to see how they effect the paint times. This gives us much finer-grained insight into the paint profile of the page.

Here, we see that Pinterest is using `box-shadow`<sup>217</sup> on these pins. We’ve optimized the performance of `box-shadow` in Chrome over the past two years, but in combination with other styles and when heavily used, it could cause a bottleneck, so it’s worth looking at.

Pinterest has reduced continuous paint mode times by 40% by moving `box-shadow` to a separate element that doesn’t have `border-radius`. The side effect is slightly fuzzy-looking corners; however, it is barely noticeable due to the color scheme and the low `border-radius` values.

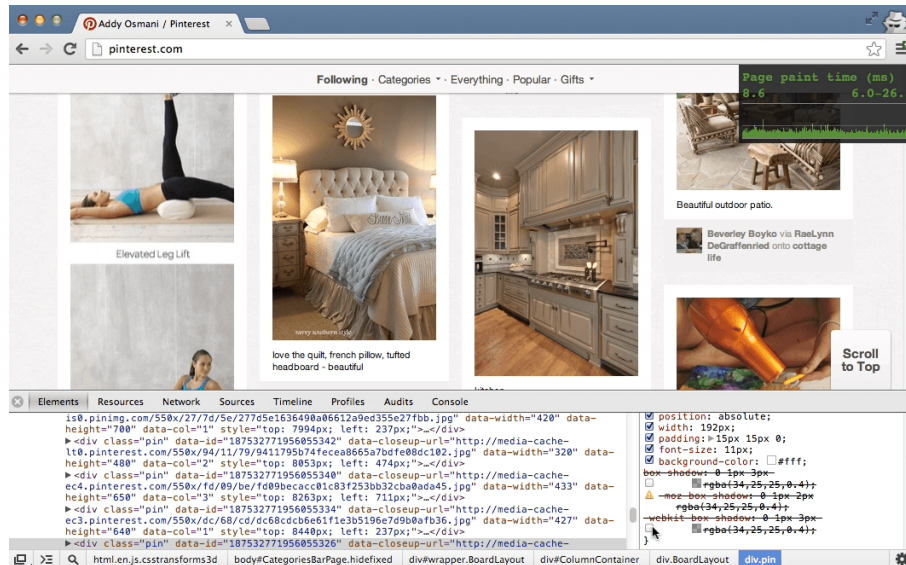
---

<sup>216</sup>. <http://media.smashingmagazine.com/wp-content/uploads/2013/05/cont.gif>

<sup>217</sup>. <http://www.html5rocks.com/en/tutorials/speed/css-paint-times/>



**Note:** You can read more about this topic in “CSS Paint Times and Page Render Weight<sup>218</sup>.”



*Toggling styles to measure their effect on page-rendering weight.  
(Animated GIF<sup>219</sup>)*

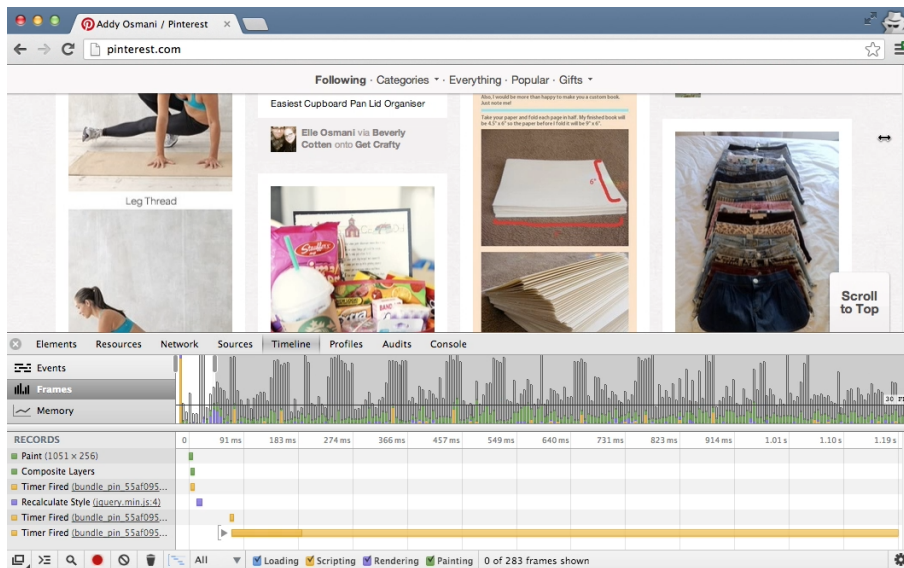
Let’s disable **box-shadow** to see whether it makes a difference. As you can see, it’s no longer visible on any of the pins. So, let’s go back to the timeline and record a new session in which we scroll the same way as we did before (up and down, up and down, up and down). We’re getting closer to 60 FPS now, and that’s just from one change.

**Public service announcement:** We’re absolutely not saying don’t use **box-shadow** — by all means, do! Just make sure that if you have a performance problem, measure correctly to find out what your own bottlenecks are.

<sup>218</sup>. <http://www.html5rocks.com/en/tutorials/speed/css-paint-times/>

<sup>219</sup>. <http://media.smashingmagazine.com/wp-content/uploads/2013/05/box.gif>

Always measure! Your website or application is unique, as will any performance bottleneck be. Browser internals change almost daily, so measuring is the smartest way to stay up to date on the changes, and Chrome's Developer Tools makes this really easy to do.



*Using Chrome Developer Tools to profile is the best way to track browser performance changes.*

**Note:** Eberhard Grather recently wrote a detailed post on “[Profiling Long Paint Times With DevTools’ Continuous Painting Mode](http://updates.html5rocks.com/2013/02/Profiling-Long-Paint-Times-with-DevTools-Continuous-Painting-Mode)”<sup>220</sup>, which you should spend some quality time with.

Another thing we noticed is that if you click on the “Re-pin” button, do you see the animated effect and the light-box being painted? There’s a big red flash of repaint in the

---

<sup>220</sup>. <http://updates.html5rocks.com/2013/02/Profiling-Long-Paint-Times-with-DevTools-Continuous-Painting-Mode>



background. It's not clear from the tooling if the paint is the white cover or some other affected being area. Be sure to double check that the paint rectangles correspond to the element or elements that you think are being repainted, and not just what it looks like. In this case, it looks like the whole screen is being repainted, but it could well be just the white cover, which might not be all that expensive. It's nuanced; the important thing is to understand what you're seeing and why.

## **HARDWARE COMPOSITING (GPU ACCELERATION)**

The last thing we're going to look at on Pinterest is GPU acceleration. In the past, Web browsers have relied pretty heavily on the CPU to render pages. This involved two things: firstly, painting elements into a bunch of textures, called layers; and secondly, compositing all of those layers together to the final picture seen on screen.

Over the past few years, however, we've found that getting the GPU involved in the compositing process can lead to some significant speeding up. The premise is that, while the textures are still painted on the CPU, they can be uploaded to the GPU for compositing. Assuming that all we do on future frames is move elements around (using CSS transitions or animations) or change their opacity, we simply provide these changes to the GPU and it takes care of the rest. We essentially avoid having to give the GPU any new graphics; rather, we just ask it to move existing ones around. This is something that the GPU is

exceptionally quick at doing, thus improving performance overall.

There is no guarantee that this hardware compositing will be available and enabled on a given platform, but if it is available the first time you use, say, a 3D transform on an element, then it will be enabled in Chrome. Many developers use the `translateZ` hack to do just that. The other side effect of using this hack is that the element in question will get its own layer, which may or may not be what you want. It can be very useful to effectively isolate an element so that it doesn't affect others as and when it gets repainted. It's worth remembering that the uploading of these textures from system memory to the video memory is not necessarily very quick. The more layers you have, the more textures need to be uploaded and the more layers that will need to be managed, so it's best not to overdo it<sup>221</sup>.

**Note:** Tom Wiltzius has written about the layer model in Chrome<sup>222</sup>, which is a relevant read if you are interested in understanding how compositing works behind the scenes. Paul has also written a post about the `translateZ` hack<sup>223</sup> and how to make sure you're using it in the right ways.

Another great setting in Developer Tools that can help here is "Show composited layer borders." This feature will

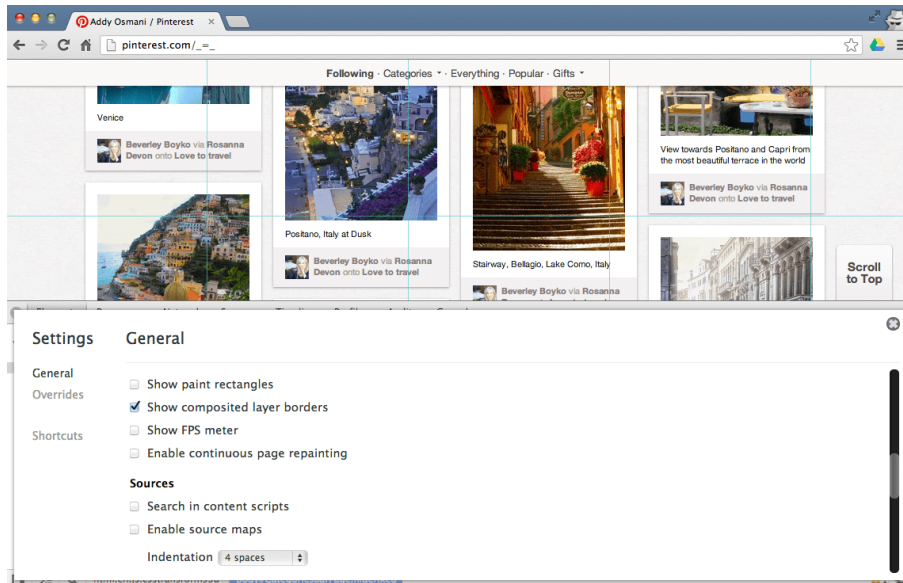
---

<sup>221</sup>. <https://plus.google.com/115133653231679625609/posts/gv92WXBBkgU>

<sup>222</sup>. <http://www.html5rocks.com/en/tutorials/speed/layers/>

<sup>223</sup>. <http://aerotwist.com/blog/on-translate3d-and-layer-creation-hacks/>

give you insight into those DOM elements that are being manipulated at the GPU level.



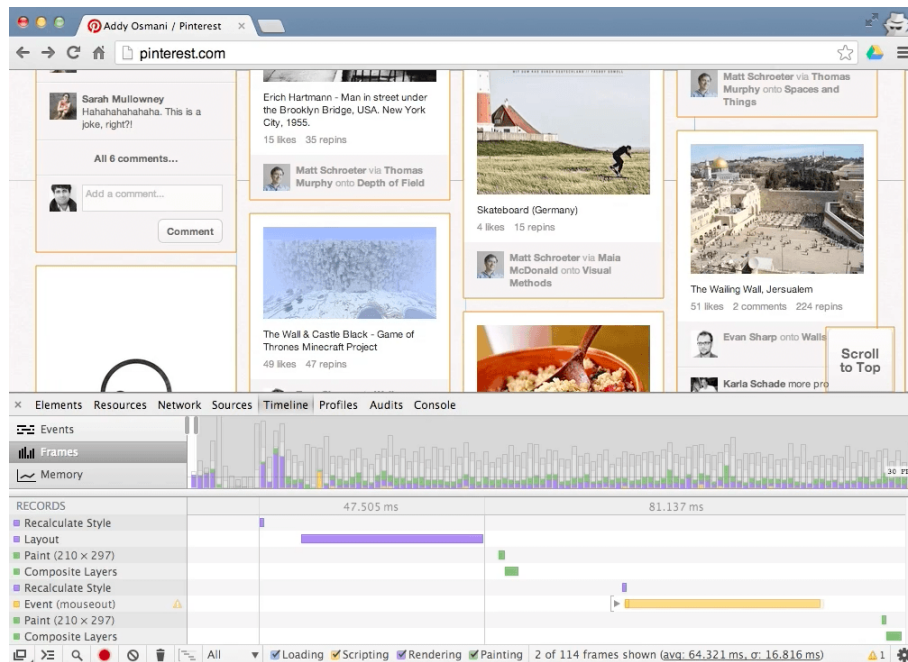
*Switching on composited layer borders will indicate Chrome's rendering layers. (Animated GIF<sup>224</sup>)*

If an element is taking advantage of the GPU acceleration, you'll see an orange border around it with this on. Now as we scroll through, we don't really see any use of composited layers on this page – not when we click “Scroll to top” or otherwise.

Chrome is getting better at automatically handling layer promotion in the background; but, as mentioned, developers sometimes use the `translateZ` hack to create a composited layer. Below is Pinterest's feed with `translateZ(0)` applied to all pins. It's not hitting 60 FPS, but it is getting closer to a consistent 30 FPS on desktop, which is actually not bad.

---

<sup>224</sup>. <http://media.smashingmagazine.com/wp-content/uploads/2013/05/nolayers.gif>



Using the `translateZ(0)` hack on all Pinterest pins. Note the orange borders. (Animated GIF<sup>225</sup>)

Remember to test on both desktop and mobile, though; their performance characteristics vary wildly. Use the timeline in both, and watch your paint time chart in Continuous Paint mode to evaluate how fast you're busting your budget.

Again, don't use this hack on every element on the page — it might pass muster on desktop, but it won't on mobile. The reason is that there is increased video memory usage and an increased layer management cost, both of which could have a negative impact on performance. Instead, use hardware compositing only to isolate elements where the paint cost is measurably high.

---

<sup>225</sup>. <http://media.smashingmagazine.com/wp-content/uploads/2013/05/transformpost.gif>

**Note:** In the WebKit nightlies<sup>226</sup>, the Web Inspector now also gives you the reasons<sup>227</sup> for layers being composited. To enable this, switch off the “Use WebKit Web Inspector” option and you’ll get the front end with this feature in there. Switch it on using the “Layers” button.

## *A Find-and-Fix Workflow*

Now that we’ve concluded our Pinterest case study, what about the workflow for diagnosing and addressing your own paint problems?

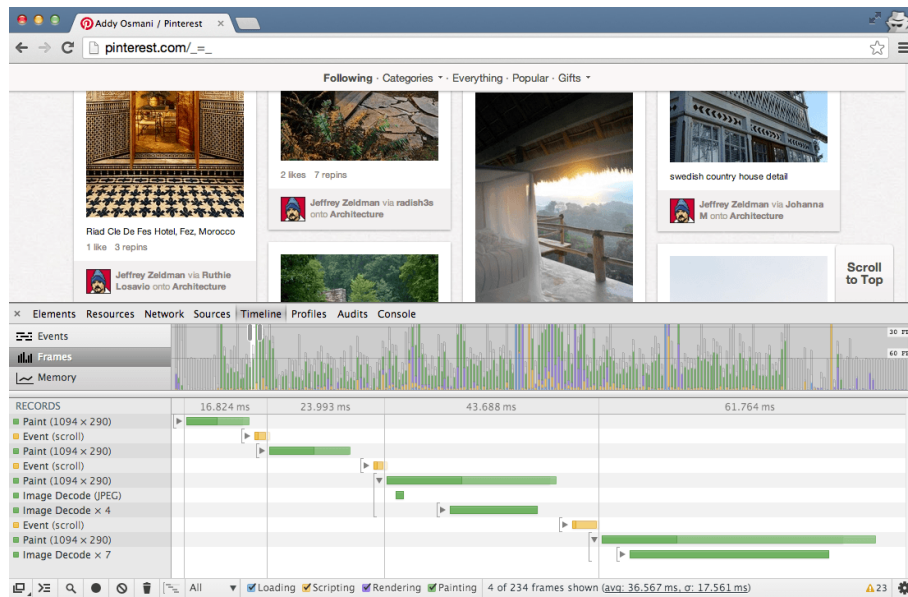
### FINDING THE PROBLEM

- Make sure you’re in “Incognito” mode. Extensions and apps can skew the figures that are reported when profiling performance.
- Open the page and the Developer Tools.
- In the timeline, record and interact with your page.
- Check for frames that go over budget (i.e. over 60 FPS).
- If you’re close to budget, then you’re likely way over the budget on mobile.
- Check the cause of the jank. Long paint? CSS layout? JavaScript?

---

<sup>226</sup>. <http://nightly.webkit.org/>

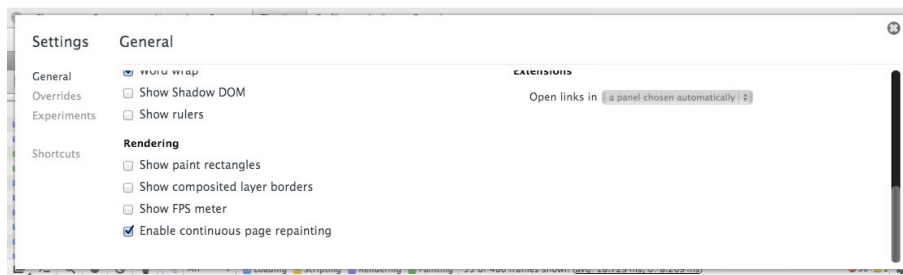
<sup>227</sup>. <https://twitter.com/addyosmani/status/313978378220879872/photo/1>



*Spend some quality time with Frame mode in Chrome Developer Tools to understand your website's runtime profile.*

## FIXING THE PROBLEM

- Go to “Settings” and enable “Continuous Page Repainting.”
- In the “Elements” panel, hide anything non-essential using the hide (H) shortcut.
- Walk through the DOM tree, hiding elements and checking the FPS in the timeline.
- See which element(s) are causing long paints.
- Uncheck styles that could affect paint time, and track the FPS.
- Continue until you’ve located the elements and styles responsible for the slow-down.



*Switch on extra Developer Tools features for more insight.*

## What About Other Browsers?

Although at the time of writing, Chrome has the best tools to profile paint performance, we strongly recommend testing and measuring your pages in other browsers to get a feel for what your own users might experience (where feasible). Performance can vary massively between them, and a performance smell in one browser might not be present in another.

As we said earlier, don't guess it, test it! Measure for yourself, understand the abstractions, know your browser's internals. In time, we hope that the cross-browser tooling for this area improves so that developers can get an accurate picture of rendering performance, regardless of the browser being used.

## Conclusion

Performance is important. Not all machines are created equal, and the fast machines that developers work on might not have the performance problems encountered on the devices of real users. Frame rate in particular can have a big impact on engagement and, consequently, on a

project's success. Luckily, a lot of great tools out there can help with that.

Be sure to measure paint performance on both desktop and mobile. If all goes well, your users will end up with snappier, more silky-smooth experiences, regardless of the device they're using.

## FURTHER READING

- [“Performance Profiling With the Timeline<sup>228</sup>,”](#) Chrome DevTools, Google Developers
- [Let's Make the Web Jank-Free<sup>229</sup>](#) (resources)
- [“Don't Guess It, Test It!<sup>230</sup>”](#) (article and video), Paul Lewis
- [“CSS Paint Times and Page Render Weight<sup>231</sup>,”](#) Colt McAnlis, HTML5 Rocks
- [“Accelerated Rendering in Chrome<sup>232</sup>,”](#) Tom Wiltzius, HTML5 Rocks
- [“Avoiding Unnecessary Paints<sup>233</sup>,”](#) Paul Lewis, HTML5 Rocks
- [“Fluid User Interface With Hardware Acceleration<sup>234</sup>”](#) (slidedeck) Ariya Hidayat, W3Conf 2013 🐼

---

<sup>228</sup>. <https://developer.chrome.com/devtools/docs/timeline>

<sup>229</sup>. <http://jankfree.org>

<sup>230</sup>. <http://aerotwist.com/blog/dont-guess-it-test-it/>

<sup>231</sup>. <http://www.html5rocks.com/en/tutorials/speed/css-paint-times/>

<sup>232</sup>. <http://www.html5rocks.com/en/tutorials/speed/layers/>

<sup>233</sup>. <http://www.html5rocks.com/en/tutorials/speed/unnecessary-paints/>

<sup>234</sup>. <https://speakerdeck.com/ariya/fluid-user-interface-with-hardware-acceleration>



# About The Authors

## Addy Osmani

Addy Osmani is a Developer Programs Engineer on the Chrome team at Google. A passionate JavaScript developer, he has written open-source books like *Learning JavaScript Design Patterns*<sup>235</sup> and *Developing Backbone Applications*<sup>236</sup>, having also contributed to open-source projects like Modernizr and jQuery. He is currently working on ‘Yeoman’ – an opinionated workflow for building beautiful applications. Twitter: [@addyosmani](https://twitter.com/addyosmani)<sup>237</sup>.

## Bobby Pearson

Bobby Pearson is a web programmer for [The Ivy Group](http://www.ivygroup.com)<sup>238</sup> in Charlottesville, Virginia. He began his professional life as a .NET application developer for his alma mater, the University of Virginia, and pivoted into open-source web technologies for a more dynamic and varied work experience. Bobby has been the lead programmer on dozens of websites and loves it when a plan comes together. Valid HTML, clever JavaScript, clean CSS, and neatly refactored PHP give him a cosmic sense of satisfaction. Bobby lives with his wife Audrey and their two children Luke and Lydia just down the street from Thomas Jefferson’s Monticello. In season, he brings the fury as a linebacker on the

---

<sup>235</sup>. <http://shop.oreilly.com/product/0636920025832.do>

<sup>236</sup>. <http://shop.oreilly.com/product/0636920025344.do>

<sup>237</sup>. <http://www.twitter.com/addyosmani>

<sup>238</sup>. <http://www.ivygroup.com>

Blue Ridge Church of Christ flag football team. Out of season, he makes a mean spicy chili.

## *James Rosewell*

James Rosewell<sup>239</sup> has 18 years experience in the mobile industry. He led many initiatives for Vodafone, including the development of the 1st mobile web application for the Nokia 7110 in 1999. Since then he's founded several businesses, including 51Degrees, providers of the fastest and most accurate device detection tools<sup>240</sup> tools to build on RWD techniques<sup>241</sup>. Licensed under the Mozilla Public Licence 2 all source code is freely available for commercial use. Between 150 and 350 new web enabled devices<sup>242</sup> are added to the 51Degrees data set each week by a professional team of over 9 people. The extensive information available ranges from screen size<sup>243</sup>, retail price<sup>244</sup> and battery capacity<sup>245</sup> to browser capabilities<sup>246</sup>. James is also a regular presenter on thefonecast.com<sup>247</sup>, the longest running mobile industry podcast. Twitter: @jwrosewell<sup>248</sup>.

---

<sup>239</sup>. <http://jamesrosewell.com/>

<sup>240</sup>. <https://51degrees.com/Products/Device-Detection>

<sup>241</sup>. <https://51degrees.com/Products/Responsive-Web-Design>

<sup>242</sup>. <https://51degrees.com/Support/Documentation/Device-Data>

<sup>243</sup>. <https://51degrees.com/Resources/Property-Dictionary#Screen>

<sup>244</sup>. <https://51degrees.com/Resources/Property-Dictionary#Price>

<sup>245</sup>. <https://51degrees.com/Resources/Property-Dictionary#Battery>

<sup>246</sup>. <https://51degrees.com/Resources/Property-Dictionary#BrowserUA>

<sup>247</sup>. <http://thefonecast.com/>

<sup>248</sup>. <https://twitter.com/jwrosewell>

## Johan Johansson

Johan Johansson<sup>249</sup> is a Web Development Manager at Pixelmade<sup>250</sup> in Vancouver, Canada. He has founded two web design companies during his 18 year career. His free time is consumed by his 4 year old son, who won't take no for an answer. You can follow Johan on Twitter [@johansson\\_johan](https://twitter.com/johansson_johan)<sup>251</sup>.

## Marcus Taylor

Marcus Taylor is the founder of Venture Harbour<sup>252</sup>, a digital marketing agency that specialises in working with companies in the music, film, and game industries.

Twitter: [@MarcusATaylor](https://twitter.com/MarcusATaylor)<sup>253</sup>. Google Profile: <https://plus.google.com/+MarcusTaylorVH><sup>254</sup>.

## Maximiliano Firtman

Maximiliano Firtman is a mobile and web developer, consultant, trainer and author of the books *Programming the Mobile Web*<sup>255</sup> (2nd edition) and *Up and Running: jQuery Mobile*<sup>256</sup>. He is a frequent speaker, including Fluent, JSConf, TopConf, Velocity Conference talking about performance and mobile HTML5. He has delivered more than 200

---

<sup>249</sup>. <https://plus.google.com/112633905866117636066?rel=author>

<sup>250</sup>. <http://www.pixelmade.com>

<sup>251</sup>. [https://twitter.com/johansson\\_johan](https://twitter.com/johansson_johan)

<sup>252</sup>. <http://www.ventureharbour.com/>

<sup>253</sup>. <http://www.twitter.com/MarcusATaylor>

<sup>254</sup>. <https://plus.google.com/+MarcusTaylorVH?rel=author>

<sup>255</sup>. <http://firt.mobi/pmw>

<sup>256</sup>. <http://firt.mobi/jqm>

trainings in more than 20 countries. He maintains the *HTML5 Mobile Compatibility list*<sup>257</sup> and usually publishes information on new capabilities on mobile browsers in his blog<sup>258</sup>. Twitter: @firt<sup>259</sup>.

## *Per Buer*

Per Buer is the CTO and founder of Varnish Software, the company behind the open source project Varnish Cache. Buer is a former programmer turned sysadmin, then manager turned entrepreneur. Runs, cross country skis and tries to keep his two boys from tearing down the house. Twitter: @perbu<sup>260</sup>.

## *Rachel Andrew*

Rachel Andrew is a web developer, writer and speaker and one of the people behind the content management system, Perch<sup>261</sup>. She is the author of a number of books including chapters in the Smashing Book #3 and Smashing Book #4<sup>262</sup>, where she writes about Providing Technical Support. She writes about business and technology on her own site at rachelandrew.co.uk<sup>263</sup>. Twitter: @rachelandrew<sup>264</sup>.

---

<sup>257</sup>. <http://mobilehtml5.org>

<sup>258</sup>. <http://www.mobilexweb.com>

<sup>259</sup>. <http://www.twitter.com/firt>

<sup>260</sup>. <http://www.twitter.com/perbu>

<sup>261</sup>. <http://grabaperch.com>

<sup>262</sup>. <https://shop.smashingmagazine.com/smashing-book-4-new-perspectives-on-web-4-design.html>

<sup>263</sup>. <http://rachelandrew.co.uk>

## Vitaly Friedman

Vitaly Friedman loves beautiful content and doesn't like to give in easily. Vitaly is writer, speaker, author and editor-in-chief of Smashing Magazine. He runs responsive Web design workshops<sup>265</sup>, online workshops<sup>266</sup> and loves solving complex performance problems in large companies. Get in touch<sup>267</sup>. Twitter: @smashingmag<sup>268</sup>.

---

<sup>264</sup>. <http://www.twitter.com/rachelandrew>

<sup>265</sup>. <https://shop.smashingmagazine.com/workshop-responsive-design-vitaly-zurich-responsiveday.html>

<sup>266</sup>. <http://www.smashingmagazine.com/smashing-workshops/#online-workshops>

<sup>267</sup>. <http://www.smashingmagazine.com/workshops/#in-house-workshop>

<sup>268</sup>. <http://www.twitter.com/smashingmag>

## *About Smashing Magazine*

Smashing Magazine<sup>269</sup> is an online magazine dedicated to Web designers and developers worldwide. Its rigorous quality control and thorough editorial work has gathered a devoted community exceeding half a million subscribers, followers and fans. Each and every published article is carefully prepared, edited, reviewed and curated according to the high quality standards set in Smashing Magazine's own publishing policy<sup>270</sup>.

Smashing Magazine publishes articles on a daily basis with topics ranging from business, visual design, typography, front-end as well as back-end development, all the way to usability and user experience design. The magazine is—and always has been—a professional and independent online publication neither controlled nor influenced by any third parties, delivering content in the best interest of its readers. These guidelines are continually revised and updated to assure that the quality of the published content is never compromised. Since its emergence back in 2006 Smashing Magazine has proven to be a trustworthy online source.

---

<sup>269</sup>. <http://www.smashingmagazine.com>

<sup>270</sup>. <http://www.smashingmagazine.com/publishing-policy/>