Page Weight Doesn't Matter

by **Nate Berkopec** (@nateberkopec) of ***speedshop*** (who?), a Rails performance consultancy.

Summary: The total size of a webpage, measured in bytes, has little to do with its load time. Instead, increase network utilization: make your site preloader-friendly, minimize parser blocking, and start downloading resources ASAP with Resource Hints. *(4697 words/23 minutes)*

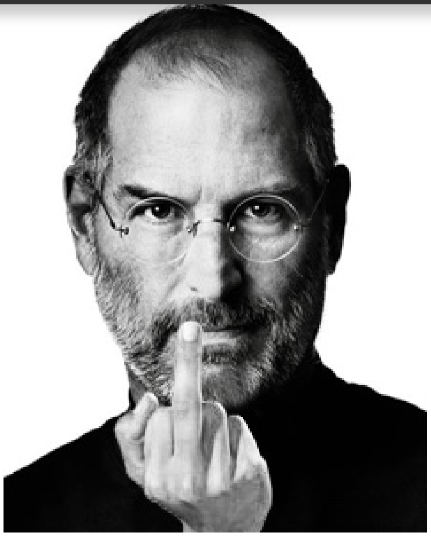
There's one universal law of front-end performance - **less is more**. Simple pages are fast pages. We all know this - it isn't controversial. Complexity is the enemy.

And yet, it's trivial to find a website whose complexity seems to reach astronomical levels. [1] It's perhaps telling that media and news sites tend to be the worst here - most media sites in 2015 take ages to load, not to mention all the time you spend clicking past their paywall popups (NYTimes) or full-page advertisements (Forbes).

[1] Literally. The Apollo Guidance Computer had just 64 KB of ROM, but most webpages require more than 1MB of data to render. There are some webpages that are actually 100x as complex as the software that took us to the moon.

Remember when Steve Jobs said A
products would never support Flasl
there, it was a bit of a golden age i
broadband was becoming widespre
come on the scene, and, most impo

started dropping Flash cruft. The "loading!" screens and unnecessarily complicated navigation schemes became something of yesteryear.


"Dear Adobe: Flash is a dumpster fire. Love, Steve."

That, is, until the marketing department figured out how to use Javascript. The Guardian's homepage sets advertising tracking cookies across 4 different partner domains. Business Insider thought to one-up their neighbors across the pond and sets **cookies across 17 domains**, requires **284 requests** (to nearly 100 unique domains) and a **4.9MB download** which took a full *9 seconds* to load on my cable connection, which is a fairly average broadband ~20 megabit pipe. Business Insider is, ostensibly, a news site. The purpose of the Business Insider is to deliver text content. Why does that require 5 MB of *things which are not text*?

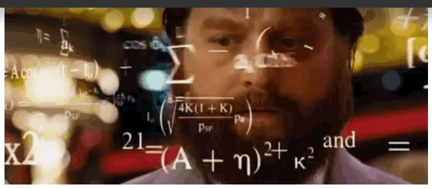
"They think Business Insider is a news site and not just an ad delivery mechanism? That's rich!"

Unfortunately, it seems, the cry of "complexity is the enemy!" is lost on the ones setting the technical agenda. While trying to load every single tracking cookie possible on your users, you've steered them away by making your site slow on *any* reasonable broadband connection, and nearly *impossible* on any mobile connection.

Usually, the boogeyman that gets pointed at is *bandwidth*: users in low-bandwidth
developing world) are getting shaft

But the math doesn't *quite* work ou
global connection speed average at

second. So wait a second - why does Business Insider take 9 seconds to load on my 20 megabit pipe, when it's only 4.9MB? If I had an average connection, according to Akamai, shouldn't Business Insider load in 2 seconds, tops?



4 divided by 20 isn't 9…

The secret is that "page weight", broadly defined as the simple total file size of a page and all of it's sub-resources (images, CSS, JS, etc), isn't the problem. **Bandwidth is not the problem, and the performance of the web will not improve as broadband access becomes more widespread.**

The problem is latency.

Most of our networking protocols require a lot of round-trips. Each of those round trips imposes a latency penalty. Latency is governed, at the end of the day, by the speed of light. Which means that latency *isn't going anywhere*.

DNS lookup is, and always will be, expensive.[2]

[2] I'm being facetious, of course. In 10 years, we may have invented some better protocols here. But it's fair to say we have to live with the current reality for at least a decade. Look at how long it's taking us to get on board with IPv6.

TCP connections are, and always will be, expensive.

SSL handshakes are, and always will be, expensive. We're going to be doing more of th
years. Thanks NSA.

Each of these things requires at lea
trip - that is, a packet going from y

the network, to someone else's. That will never be faster than the speed of light - and even light takes 30 milliseconds to go from New York to San Francisco and back. [3] What's worse is that these network round-trips must happen sequentially - we have to know the IP address before we start the three-way handshake for TCP, and we have to establish a TCP connection before we can start to negotiate SSL.

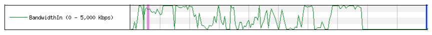
[3] Thanks to the amount of hops a packet has to make across the internet backbone, usually the time is much worse - 2-4x.

Setting up a typical HTTPS connection can involve *5.5 round-trips*. That's like 165 milliseconds [4] per connection *on a really really good day*.

[4] In the hypothetical NY-to-SF scenario. Usually it's better than this in the US because of CDNs. But 150ms per connection isn't a bad rule of thumb - and on mobile it's much worse, closer to 300.

The smart ones among you may already see the solution - well, Nate, 165 milliseconds per connection isn't a problem! We'll just parallelize the connections! Boom! 100 connections opened in 165 milliseconds!

The problem is that HTML *doesn't work this way by default*.



Business Insider's network utilization over time - hardly pegged at 100%.

We'd like to imagine that the way a webpage loads is this:

1. Browser opens connection to yoursite.com, does DNS/TCP/SSL setup.
2. Browser downloads the document
3. As soon as the browser is done dov document, the browser starts down document's sub resources *at the sa*

4. Browser parses the document and fills in the necessary sub resources once they've been downloaded.

Here's what actually happens:

1. Browser opens connection to yoursite.com, does DNS/TCP/SSL setup.
2. Browser downloads the document (HTML).
3. Browser starts parsing the document. When the parser encounters a subresource, it opens a connection and downloads it. If the subresource is an external script tag, the parser stops, waits until it the script has downloaded, executes the entire script, and then moves on.
4. As soon as the parser stops and has to wait for an external script to download, it sends ahead something called a *preloader*. The preloader *may* notice and begin downloading resources *if* it understands how to (hint: a very popular Javascript pattern prevents this).

Parse the document? Nah man, I'm gonna wait for this script to download and execute.

Thanks to these little wrinkles, web page loads often have new connections opening *very* late in a page load - right before the end even! Ideally, the browser would open all of those connections like in our first scenario - immediately after the document is downloaded. We want to maximize network utilization across the life of the webpage load process.

There's four ways to accomplish th

- **Don't stop the parser.**
- **Get out of the browser preloader**

- **Utilize HTTP caching - but not *too* much**.
- **Use the Resource Hint API**.

Glossary

I’m going to use a couple of terms here and I want to make sure we’re all on the same page.

- Connection - A “connection” is one TCP connection between a client (your browser) and a server. These connections can be re-used across multiple requests through things like keep-alive.
- Request - A browser “requests” resources via HTTP. 99% of the time when we’re talking about requesting resources, we’re talking about an HTTP GET. Each request needs to use a TCP connection, though not necessarily a unique or new one (see keep-alive).
- Subresource - In browser parlance, a subresource is generally any resource required to completely load the main resource (in this case, the document). Examples of subresources include external Javascript (that is, `script` tags with a `src` attribute), external CSS stylesheets, images, favicons, and more.
- Parser - When a browser tries to load your webpage, it uses a parser to read the document
resources need to be fetched and to
The parser is responsible for gettin
of the first important events during
DOMContentLoaded.

Letting the Preloader do it's Job

Sometimes the parser has to stop and wait for an external resource to download - 99% of the time, this is an external script. When this happens, the browser starts something called a preloader. The preloader is a bit like a "parser-lite", but rather than construct the DOM, the preloader is more like a giant regex that searches for sub resources to download. If it finds a subresource (say an external script at the end of the document), it will start downloading it *before* the parser gets to it.

You may be thinking this is rather ridiculous - why should a browser stop completely when it sees an external script tag? Well, thanks to The Power of Javascript, that external script tag *could* potentially wreak havoc on the document if it wanted. Heck, it could completely erase the entire document and start over with `document.write()`. The browser just doesn't know. So rather than keep moving, it has to wait, download, and execute. [5]

[5] All in the HTML spec.

Browser preloaders were a huge in
performance when they arrived on
unoptimized webpages could speed
just thanks to the preloader fetching

That said, there are ways to help the preloader and there are ways to hinder it. We want to help the preloader as much as possible, and sometimes we want to stay the hell out of it's way.

Stop inserting scripts with "async" script-injection

The marketing department says you need to integrate your site with SomeBozoAdService. They said it's really easy - you just have to "add five lines of code!". You go to SomeBozoAdService's developer section, and find that they tell you to insert this into your document somewhere:

It's just one more script tag!

```
var t = document.createElement('script');
t.src = "//somebozoadservice.com/ad-tracker.js";
document.getElementsByTagName('head')
[0].appendChild(script);
```

There are other problems with this pattern (it blocks page rendering until it's done, for one), but here's one really important one - browser preloaders can't work with this. Preload scanners are *very* simple - they're simple so that they can be fast. And when they see one of these async-injected scripts, they just give up and move on. So your browser can't download the resource until the main parser thread gets to
better to use `async` and `defer`
script tags instead, to get this:

```
<script src="//somebozoadservice.com/ad-tracker.js"
async defer></script>
```

Kaboom! There are some other advantages to `async` that I get into in this other post here, but be aware that one of them is that the browser preloader can get started downloading this script before the parser even gets there.

Here's a list of other things that generally don't work with browser preloaders:

- IFrames. Sometimes there's no way around using an iframe, but if you have the option - try not to. The content of the frame can't be loaded until the parser gets there.
- @import. I'm not sure of anyone that uses @import in their production CSS, but don't. Preloaders can't start fetching `@import`ed stylesheets for you.
- Webfonts. Here's an interesting one. I could write a whole article on webfont speed (I should/will!), but they usually aren't preloaded. This is fixable with resource hints (we'll get to that in a second).
- HTML5 audio/video. This is also fixable with resource hints.

Design department: "But we need these 90 fonts to spice up the visual interest of the page!"

I've heard that in the past, preloaders wouldn't scan the body tag when blocked in the head
it is no longer true in Webkit based

In addition, modern preloaders are
request resources that are already c

HTTP caching…

HTTP caching

The fastest HTTP request is the one that is never made. That's really all HTTP caching is for - preventing unnecessary requests. Cache control headers are really for telling clients "Hey - this resource, it's not going to change very quickly. Don't ask me again for this resource until…" That's awesome. We should do that everywhere possible.

Yet, the size of the resource cache is smaller than you might think. Here's the default disk cache size in modern browsers:

Browser	Cache Size (default)
Internet Explorer 9+	~250MB
Chrome	200MB
Firefox	540MB
Mobile Safari	0
Android (all)	~25-80

Not as large as you might imagine. right - Mobile Safari does not have cache.

Most browser resource caches work on an LRU basis - last recently used. So if something doesn't get used in the cache, it's the first thing to be evicted if the cache fills up.

A pattern I've often seen is to use 3rd-party, CDN-hosted copies of popular libraries in an attempt to leverage HTTP caching. The idea is to use Google's copy of JQuery (or what have you), and a prospective user to your site will already have it downloaded before coming to yours. The browser will notice it's already in their cache, and not make a new request. There's some other benefits, but I want to pick on this one.

This sounds good in theory, but given the tiny size of caches, I'm not sure if it really works in practice. Consider how few sites actually use Google-hosted (or Cloudflare-hosted, or whatever) JQuery. Even if they did - how often is your cached copy pushed *out* of the cache by other resources? Do you know?

Consider the alternative - bundling JQuery into your application's concatenated "application.js" file (Rails' default behavior).

In the best case, the user already has the 3rd-party CDN-hosted JQuery downloaded and cac
go and get your application.js does
because it's ~20kb smaller now tha
JQuery. But remember what we sai
is hardly the issue for most connec

really saving <100ms, even on a 2MB/s DSL connection).

But consider the worst case scenario - the user doesn't have our 3rd-party JS downloaded already. Now, compared to the "stock" application.js scenario, you have to make an additional new connection to a new domain, likely requiring SSL/TLS negotiation. Without even downloading the script, you've been hit with 1-300ms of network latency. Bummer.

Consider how much worse this gets when you're including more than 1 library from an external CDN. God forbid that the script tags aren't `async`, or your user will be sitting there for a while.

In conclusion, 3rd-party hosted Javascript, while a good idea and, strictly speaking, faster in the best-case scenario, is likely to impose a huge performance penalty to users that don't have every single one of your 3rd-party scripts cached already. Far preferable is to bundle it into a single "application.js" file, served from your own domain. That way, we can re-use the already warm connection (as long you allowed the browser to "keep-alive" the connection it used to download the document) to download all of your external Javascript in one go.

Resource hints

There's another way we can maximize network utilization - through something called *resource hints*. There are couple of different kinds of resource hints. In general, most of them are telling the browser to *prepare some connection or resource in advance* of the parser getting to the actual point where it needs the connection. This prevents the parser from blocking on the network.

- **DNS Prefetch** - Pretty simple - tell the browser to resolve the DNS of a given hostname (`example.com`) as soon as possible.
- **Preconnect** - Tells the browser to open a connection as soon as possible to a given hostname. Not only will this resolve DNS, it will start a TCP handshake and perform TLS negotiation if the connection is SSL.
- **Prefetch** - Tells to browser to download an entire resource (or subresource) that may be required later on. This resource can be an entire HTML document (for example, the next page of search results), or it can be a script, stylesheet, or other subresource. The resource is only downloaded - it isn't parsed (if script) or rendered (if HTML).
- **Prerender** - One of these things is not like the other, and prerender is it. Marking an `<a>` tag with `prerender` will actually cause the browser to get the linked `href` page and *render it before the user even clicks the anchor!* This is the technology beh
Pages and Facebook's Instant Artic

It's important to note that all of the
browser may or may not act upon t

time, though, they will - and we can use this to our advantage.

Browser support: I've detailed which browsers support which resource hints (as of November 2015) below. However, any user agent that doesn't understand a particular hint will just skip past it, so there's no harm in including them. Most resource hints enjoy >50% worldwide support (according to to caniuse.com) so I think they're definitely worth including on any page.

Let's talk about each of these items in turn, and when or why you might use each of them:

DNS Prefetch

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

In case you're brand new to networking, here's a review - computers don't network in terms of domain names. Instead, they use IP addresses (like `192.168.1.1`, etc). They *resolve* a hostname, like `example.com`, into an IP address. To do this, they have to go to a DNS server (for example, Google's serv
ask: "Hey, what's the IP address of
`some-host.com`?" This connec
usually somewhere between 50-10

take much longer on mobile networks or in developing countries (500-750ms).

When to Use It: But you may be asking - why would I ever want to resolve the DNS for a hostname and *not actually connect to that hostname*? Exactly. So forget about `dns-prefetch`, because it's cousin, `preconnect`, does exactly that.

"Stop trying to make dns-prefetch a thing!"

Browser Support: Everything except IE 9 and below.

Preconnect

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

A preconnect resource hint will hint the browser to do the following:

- Resolve the DNS, if not done already (1 round-trip)
- Open a TCP connection (1.5 round-trips)
- Complete a TLS handshake if the connection is HTTPS (2-3 round-trips)

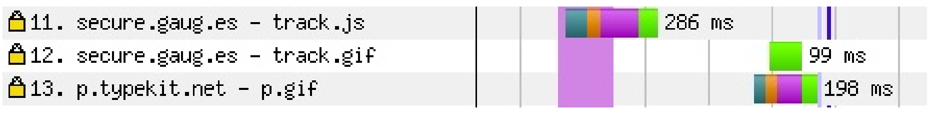
The only thing it won't do is actual
(sub)resource - the browser won't s
resource until either the parser or p
download the resource. This can el
trips across the network! That can s

of time in most environments, even fast home Wifi connections.

When to Use It: Here's an example from Rubygems.org.

Taking a look at how Rubygems.org loads in webpagetest.org, we notice a few things. What we're looking for is network utilization after the document is downloaded - once the main "/" document loads, we should see a bunch of network requests fire at once. Ideally, they'd all fire off at this point. In a perfect world, network utilization would look like a flat line at 100%, which then stops as soon as the page loads completely. Preconnect helps us to do that by allowing us to move some network tasks earlier in the page load process.

Notice these these two resources, closer to the end of the page load:


```
11. secure.gaug.es - track.js   286 ms
12. secure.gaug.es - track.gif  99 ms
13. p.typekit.net - p.gif       198 ms
```

Two are related to gaug.es, an analytics tracking service, and the other is a GIF from a Typekit domain. The green bar here is time-to-first-byte - time
server response. But note how the a
service and the Typekit GIF have t
purple bars as well - these bars rep
resolving DNS, opening a connecti

SSL, respectively. By adding a preconnect tag to the head of the document, we can move this work to the beginning of the page load, so that when the browser needs to download these resource it has a pre-warmed connection. That loads each resource ~200ms faster in this case.

You may be wondering - why hasn't the preloader started loading these resources earlier? In the case of the gang.es script, it was loaded with an "async" script-injection tag. This is why that method is a bit of a stinker. For more about why script-injection isn't a great idea, see Ilya Grigorik's post on the topic. So in this case, rather than adding a `preconnect` tag, I'll simply change the gaug.es script to a regular script tag with an `async` attribute. That way, the browser preloader will pick it up and download it as soon as possible.

In the case of that Typekit gif, it was also script-injected into the bottom of the document. A `preconnect` tag would speed up this connection. However, `p.gif` is actually a tracking beacon for Adobe, so I don't think that speeding that up will provide any performance benefit to the user.

In general, `preconnect` works
resources that are script-injected, b
preloader cannot download these re
webpagetest.org to seek out sub res
and trigger the DNS/TCP/TLS setu

In addition, it works very well for script-injected resources with dynamic URLs. You can set up a connection to the domain, and then later use that connection to download a dynamic resource (like the Typekit example above). See the W3C spec:

> The full resource URL may not be known until the page is being constructed by the user agent - e.g. conditional loading logic, UA adaptation, etc. However, the origin from which one or more of these resources will be fetched is often known ahead of time by the developer or the server generating the response. In such cases, a preconnect hint can be used to initiate an early connection handshake such that when the resource URL is determined, the user agent can dispatch the request without first blocking on connection negotiation.

Browser Support: Unfortunately, preconnect is probably the least-supported resource hint. It only works in *very* modern Chrome and Firefox versions, and is coming to Opera soon. Safari and IE don't support it.

Prefetch

```
<link rel="prefetch" href="//examp
image.gif">
```

A prefetch resource hint will hint the browser to do the following:

- Everything that we did to set up a connection in the `preconnect` hint (DNS/TCP/TLS).
- But in addition, the browser will also *actually download the resource*.
- However, `prefetch` only works for resources required by *the next navigation*, not for the *current page*.

Go get the resource, Chrome! Go get it, boy!

When to Use It: Consider using `prefetch` in any case where you have a good idea what the user might do next. For example, if we were implementing an image gallery with Javascript, where each image was loaded with an AJAX request, we might insert the following prefetch tag to load the next image in the gallery:

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

You can even prefetch entire pages. Consider a paginated search result:

```
<link rel="prefetch" href="//example.com/search?
q=test&page=2">
```

Browser Support: IE 11 and up, F

Opera all support `prefetch`. Sa

don't.

Prerender

Prerender is prefetch on steroids - instead of just downloading the linked document, it will actually pre-render the entire page! Obviously, this means that pre rendering only works for HTML documents, not scripts or other subresources.

This is a great way to implement something like Google's Instant Pages or Facebook's Instant Articles.

Of course, you have to be careful and considerate when using prefetch and prerender. If you're prefetching something on your own server, you're effectively adding another request to your server load for every prefetch directive. A prerender directive can be even more load-intensive because the browser will also fetch all sub resources (CSS/JS/images, etc), which may also come from your servers. It's important to only use prerender and prefetch where you can be pretty certain a user will actually use those resources on the next navigation.

There's another caveat to prerender - like all resource hints, prerenders are given much lo

browser and aren't always execute

the spec:

"The user agent may:

- Allocate fewer CPU, GPU, or memory resources to pre rendered content.
- Delay some requests until the requested HTML resource is made visible - e.g. media downloads, plugin content, and so on.
- Prevent pre rendering from being initiated when there are limited resources available.”

Browser Support: IE 11 and up, Chrome, and Opera. Firefox, Safari and iOS Safari don’t get this one.

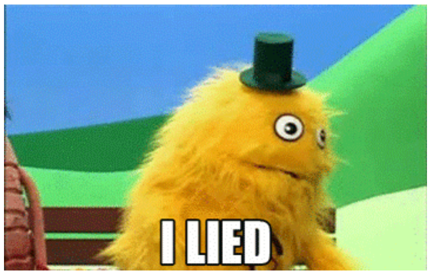
Conclusion

We have a long way to go with performance on the web. I scraped together a little script to check the Alexa Top 10000 sites and look for resource hints - here’s a quick table of what I found.

Resource Hint	Prevalence
`dns-prefetch`	5.0%
`preconnect`	0.4%
`prefetch`	
`prerender`	

So many sites could benefit from li
all of these resource hints, but so fe

do use them are just using `dns-prefetch`, which is practically useless when compared to the superior `preconnect` (how often do you really want to know the DNS resolution of a host and then *not* connect to it?).



I'd like to back off from the flamebait-y title off this article *just* slightly. Now that I've explained all of the different things you can do to increase network utilization during a webpage load, know that 100% utilization isn't always possible. Resource hints and the other techniques in this article *help* complex pages load faster, but thanks to many different constraints you may not be able to apply them in all situations. Page weight *does* matter - a 5MB page will be more difficult to optimize than a 500 KB one. What I'm really trying to say is that page weight *only sorta* matters.

I hope I've demonstrated to you that page weight - while certainly *correlated* with webpage load speed, is not the final answer. You shouldn't feel like your page is doomed to slowness because The Marketing People need you to include 8 different external ad tracking services (although you should consider quitting your job if that's the case).

TL;DR:

- Don't inject scripts.
- Reduce the number of connections
reducing page size.

- HTTP caching is great, but don’t *rely* on any particular resource being cached.
- Use resource hints - especially `preconnect` and `prefetch`.

Want a faster website?

I'm Nate Berkopec (@nateberkopec). I write online about web performance from a full-stack developer's perspective. I primarily write about frontend performance and Ruby backends. If you liked this article and want to hear about the next one, click below. I don't spam - you'll receive about 1 email per week. It's all low-key, straight from me.

Products from Spee

The Complete Guide to Rails

Performance is a full-stack performance book that gives you the tools to make Ruby on Rails applications faster, more scalable, and simpler to maintain.

LEARN MORE

The Rails Performance Workshop is the big brother to my book. Learn step-by-step how to make your Rails app as fast as possible through a comprehensive video and hands-on workshop. Available for individuals, groups and large teams.

LEARN MORE

More Posts

Announcing the Rails Performance Apocrypha

I've written a new book, compiled from 4 years of my email newsletter.

Read more

We Made Puma Faster With Sleep Sort

Puma 5 is a huge major release for the project. It brings several new experimental performance features, along with tons of bugfixes and features. Let's talk about some of the most important ones.

Read more

The Practical Effects of the GVL on Scaling in Ruby

MRI Ruby's Global VM Lock: frequently mislabeled, misunderstood and maligned. Does the GVL mean that Ruby has no concurrency story or CaN'T sCaLe? To understand completely, we have to dig through Ruby's Virtual Machine, queueing theory and Amdahl's Law. Sounds simple, right?

Read more

The World Follows Power Laws: Why Premature Optimization is Bad

Programmers vaguely realize that 'premature optimization is bad'. But what is premature optimization? I'll argue that any optimization that does not come from observed measurement, usually in production, is premature, and that this fact stems from natural facts about our world. By applying an empirical mindset to performance, we can...

Read more

Why Your Rails App is Slow: Lessons Learned from 3000+ Hours of Teaching

I've taught over 200 people at live workshops, worked with dozens of clients, and thousands of readers to make their Rails apps faster. What have I learned about performance work and Rails in the process? What makes apps slow? How do we make them faster?

Read more

3 ActiveRecord Mistakes That Slow Down Rails Apps: Count, Where and Present

Many Rails developers don't under
what causes ActiveRecord to actua
execute a SQL query. Let's look at
common cases: misuse of the coun
method, using where to select subs

the present? predicate. You may be causing extra queries and N+1s through the abuse of these three methods.

Read more

The Complete Guide to Rails Performance, Version 2

I've completed the 'second edition' of my course, the CGRP. What's changed since I released the course two years ago? Where do I see Rails going in the future?

Read more

A New Ruby Application Server: NGINX Unit

NGINX Inc. has just released Ruby support for their new multi-language application server, NGINX Unit. What does this mean for Ruby web applications? Should you be paying attention to NGINX Unit?

Read more

Malloc Can Double Multi-threaded Ruby Program Memory Usage

Memory fragmentation is difficult
sometimes be very easy to fix. Let'
multi-threaded CRuby programs: m

Read more

Configuring Puma, Unicorn and Passenger for Maximum Efficiency

Application server configuration can make a major impact on the throughput and performance-per-dollar of your Ruby web application. Let's talk about the most important settings.

Read more

Is Ruby Too Slow For Web-Scale?

Choosing a new web framework or programming language for the web and wondering which to pick? Should performance enter your decision, or not?

Read more

Railsconf 2017: The Performance Update

Did you miss Railsconf 2017? Or maybe you went, but wonder if you missed something on the performance fron
me fill you in!

Read more

Understanding Ruby GC through GC.stat

Have you ever wondered how the heck Ruby's GC works? Let's see what we can learn by reading some of the statistics it provides us in the GC.stat hash.

Read more

Rubyconf 2016: The Performance Update

What happened at RubyConf 2016 this year? A heck of a lot of stuff related to Ruby performance, that's what.

Read more

What HTTP/2 Means for Ruby Developers

Full HTTP/2 support for Ruby web frameworks is a long way off - but that doesn't mean you can't benefit from HTTP/2 today!

Read more

How Changing WebFonts Made Rubygems.org 10x Faster

WebFonts are awesome and here to
However, if used improperly, they
impose a huge performance penalty
this post, I explain how Rubygems.

painted 10x faster just by making a few changes to its WebFonts.

Read more

Hacking Your Webpage's Head Tags for Speed and Profit

One of the most important parts of any webpage's performance is the content and organization of the head element. We'll take a deep dive on some easy optimizations that can be applied to any site.

Read more

How to Measure Ruby App Performance with New Relic

New Relic is a great tool for getting the overview of the performance bottlenecks of a Ruby application. But it's pretty extensive - where do you start? What's the most important part to pay attention to?

Read more

Ludicrously Fast Page Loads - A Guide for Full-Stack Devs

Your website is slow, but the backe
issues on the frontend of your site?
a webpage and how to profile it at
Google's flamegraph-for-the-brows

Read more

Action Cable - Friend or Foe?

Action Cable will be one of the main features of Rails 5, to be released sometime this winter. But what can Action Cable do for Rails developers? Are WebSockets really as useful as everyone says?

Read more

rack-mini-profiler - the Secret Weapon of Ruby and Rails Speed

rack-mini-profiler is a powerful Swiss army knife for Rack app performance. Measure SQL queries, memory allocation and CPU time.

Read more

Scaling Ruby Apps to 1000 Requests per Minute - A Beginner's Guide

Most "scaling" resources for Ruby apps are written by companies with hun requests per second. What about sc for the rest of us?

Read more

Make your Ruby or Rails App Faster on Heroku

Ruby apps in the memory-restrictive and randomly-routed Heroku environment don't have to be slow. Achieve <100ms server response times with the tips laid out below.

Read more

The Complete Guide to Rails Caching

Caching in a Rails app is a little bit like that one friend you sometimes have around for dinner, but should really have around more often.

Read more

How To Use Turbolinks to Make Fast Rails Apps

Is Rails dead? Can the old Ruby web framework no longer keep up in this age of "native-like" performance? Turbolinks provides one solution.

Read more