Sample Pages

a smattering of representative pages

(note: not in sequence!)

JavaScript Performance Rocks!

by Amy Hoy & Thomas Fuchs

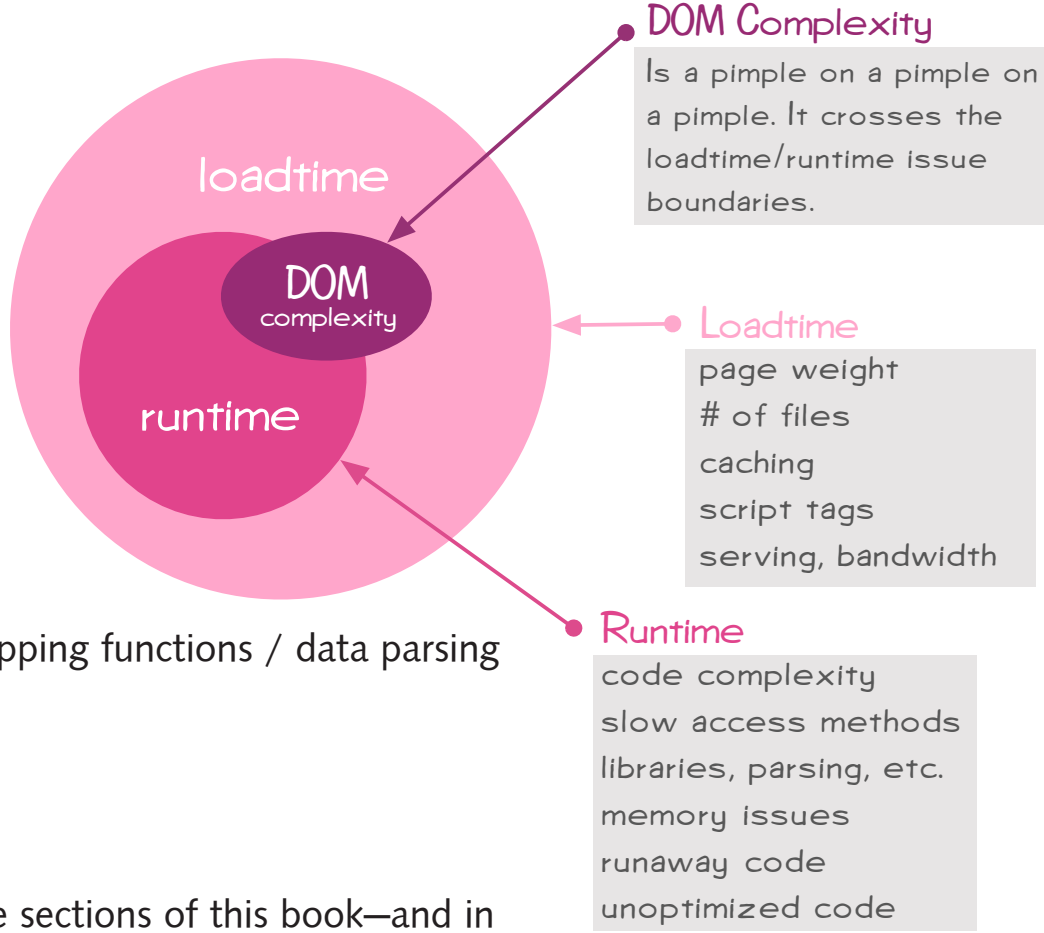
http://jsrocks.com/

Strata in the Problemosphere

The majority of rich web app issues are loadtime or runtime, and they break down something like this:

Problems in Loadtime

- page weight
- Number of files
- Caching
- DOM complexity
- <script /> practices
- serving set-up / bandwidth



Problems in Runtime

- DOM complexity
- code complexity
- slow access methods / libraries / wrapping functions / data parsing
- excessive memory use / leaks
- runaway event handlers
- unoptimized code

These two groups make up two of three sections of this book—and in that order, too. For most folks, the vast majority of problems and fixes will fall under loadtime. Loadtime issues are easy to cause, easy to find

Chapter 4

Your Friend, the DOM Monster!

Open Your Mouth & Say AaaAAGGHH!! MONSTER!

The inspiration for the DOM Monster! came from a sketch Amy made in the midst of a particularly hellish project.

The DOM Monster lives in a cave (your browser's bookmarks bar) and comes with called with a magical rite (a click). It is a mystical creature, not unlike a yeti, but rather than subsisting on rancid yak butter, it gets its jollies by traversing DOM trees.

It's a wonderful adaptation. No other creature in the universe would ever dream of stealing away this special niche.

Oh, and when it's done exploring, it gives you advice. Yeah. It's pretty awesome.

Installing the DOM Monster

Go into this book's package and check out the Goodies folder. Double-click `DOMMonster.html`. Follow the instructions. (It works in all recent browsers!)

That's it.

Naming Profile Objects & Methods

Both Safari 4 and WebKit Nightlies also have one extremely special feature: the `displayName` attribute.

Profiles in WebKit tend to be a bit confusing because so many functions used in JavaScript code tend to be anonymous or otherwise have no true name. You'll get lots of `(anonymous function)` or `(?)()` in your list.

But the guys of the Cappuccino team* have made a number of patches to the Inspector, including this one:

```
console.profile('super hero profiling');

    var rideIntoSunset = function() {
        for(i = 0; i < 1000; i++) {
          a = i * 1000;
        }
    }

    rideIntoSunset.displayName = "Ride into the sunset";
    rideIntoSunset();

console.profileEnd('super hero profiling');
```

With the `displayName` property on methods, you can control what you

*Cappucino is a development framework for desktop-like web apps, read all about it on http://objective-j.org.

Chapter 5

Firebug with YSlow

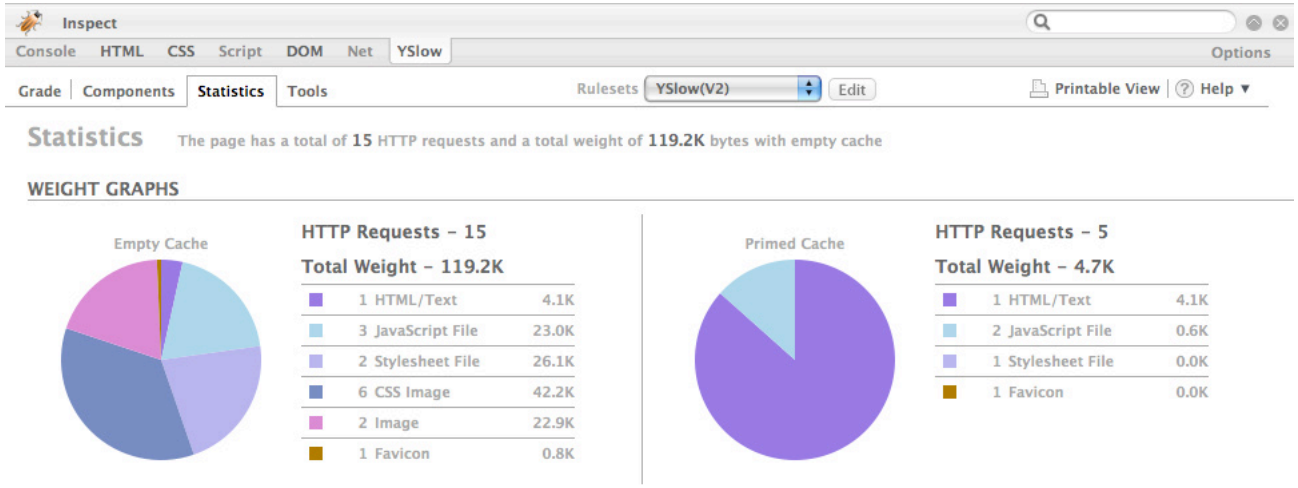
check your caching, Weigh your Assets, & Grade your Code

YSlow: the very best extension of an extension in the world!

When you're ready to take the next step, take it with YSlow.

YSlow is a Firebug plugin—and Firebug is a Firefox extension. As far as we know, there aren't any extensions for YSlow which extent Firebug which extends Firefox, but you never know.

YSlow comes from the Yahoo! performance team and is geared at one thing and one thing only: helping you check up on the performance of your web apps/pages. You get a variety of helpful tools in one package



Caching Strategies

There are two major strategies for caching JavaScript and CSS assets:

- very long cache periods (e.g. months or years; "far future cache")
- short or medium cache periods (days) ("short cache")

They've both got their pros and cons, and, unfortunately, both require a different support system to work right.

For sake of argument, let's pretend we've got this interesting little file that we want to cache and it's called `our_app.js`.

Here's how it'd work with both strategies.

Far future cache periods

With a far-future caching setup, you set the expiration date far in the future (big surprise)—months or years. That's a long time.

This seems ideal, because then your user's caches will be safe and their experience will be snappy until many happy months go by.

But meanwhile, back at the data center, you want to roll out your spiffy new psychic autocompleter—but nobody will know because you originally

Caching Strategies

There are two major strategies for caching JavaScript and CSS assets:

- very long cache periods (e.g. months or years; "far future cache")
- short or medium cache periods (days) ("short cache")

They've both got their pros and cons, and, unfortunately, both require a different support system to work right.

For sake of argument, let's pretend we've got this interesting little file that we want to cache and it's called `our_app.js`.

Here's how it'd work with both strategies.

Far future cache periods

With a far-future caching setup, you set the expiration date far in the future (big surprise)—months or years. That's a long time.

This seems ideal, because then your user's caches will be safe and their experience will be snappy until many happy months go by.

But meanwhile, back at the data center, you want to roll out your spiffy new psychic autocompleter—but nobody will know because you originally set a far-future cache expiration date to 2010.

part of their extreme tuning package.

The Closing Tag—Er, Word

Realistically speaking, there's nothing wrong with leaving off `</body></html>` or `</li>` or even `</p>`. And of course, you should self-close tags where possible.

Google, in fact, doesn't close `</body>` or `</html>` on their homepage and search results.

However, there are downsides you need to keep in mind:

- you have to use the HTML4 DTD
- you still may experience rendering issues (dropping `</body>` and `</html>` seems to cause no issues, but other tags might)
- omitting tags may increase parsing and thus rendering complexity, which means it may increase render time and runtime behaviors, even while it decreases download time

For a mini tech video on this topic, check out Google's page "Reducing the file size of HTML Documents":
http://code.google.com/speed/articles/optimizing-html.html

Chapter 5

Inlining & Precaching

Like Inline Skating, but without the scabbing

Mama always told you to separate your concerns. Or was it that there'd be days like this?

Anyway.

Common wisdom in web development circles says: separate content (HTML) from presentation (CSS), and to keep both of those faaaaar away from dirty old function (e.g. JavaScript code).

And just like every other kind of common wisdom, there are exceptions.

Brace yourself, because we're about to tell you that sometimes it makes sense to smush your CSS and JavaScript into the very same file with your HTML.

Inlining Isn't Evil

Sometimes smushing your CSS and JavaScript into one file with your HTML—called inlining—is not merely okay, but actually really beneficial.

It's natural for this to feel wrong. Just know that it's right.

Proof: Google inlines. And their whole motto is "Don't Be Evil." So it

We don't mean squishing your CSS into attributes inside your HTML tags (or JavaScript, either).

We mean inlining the whole contents of files. In the appropriate places.

Know thy Enemy

Here's how, why and when browsers will display script timeout / unresponsive page warnings:

Browser	Behavior
Internet Explorer 8	**Alert after 1 second**, prompting for user to choose to continue or cancel. Again after a further 1 second, ad infinitum.
Internet Explorer 7	**No warning**, browser just hangs until operations are done.
Safari	**Alert after 10 seconds**, prompting for user to choose to continue or cancel. Again after a further 10 seconds, ad infinitum.
Firefox	**Alert after 10 seconds**, prompting for user to choose to continue or cancel. Again after a further 10 seconds, ad infinitum.
Chrome	Alert after 20 seconds, but script continues to run. Dialog will self-close if the script finishes.

Workarounds

To fight these warnings, you have two main options:

* Reduce the problem (offload calculations to the backend; tune your approach; tune your JavaScript, etc.), or

Setting Element's Styles

Good news! If you dynamically set the styles of your elements with multiple accessors, then that means you've got some more speed you can eke out right now, without hurting too bad.

It's significantly faster to dump it all at once into the cssText accessor (just like with innerHTML). The DOM API accessors are prettier, but slower.

This is especially true if you have a huge and/or complex DOM.

Here's a couple benchmarks you can use to test it yourself:

```
benchmark(function(){
  element.style.cssText = 'font-size:'+Math.
random()*50+'px;text-indent:'+Math.random()*50+'px';
}, 10000, 'cssText');

benchmark(function(){
  element.style.fontSize = Math.random()*50+'px';
  element.style.textIndent = Math.random()*50+'px';
}, 10000, 'direct styles');
```

Even with this extremely simple example—just one DIV!—the `cssText` approach is 25% faster in Safari 3 and 50% faster in Firefox 3.

Except when it's slower...

The cssText approach is only faster when you set all the desired

Pretty == Slow?

You may have noticed a continuing theme here: "the pretty" is often also "the slow." But it doesn't have to be. With a little knowledge, careful attention, and dogged determination, beautiful visual effects can be fast. The good things in life, and on the web, are often not the same as the easy things.

Chapter 12

Method Calls Cost Money

Oh, I *can* live... with(), without you...

MCCM! It's our new chant and just the right length for a knuckle tattoo.

Method calls are expensive, so if you're hard up for cycles consider the following technique. But *only* if it doesn't make your code unmaintainable. Let's not get too hasty.

Consider inlining functionality into your loops, turning original code like this:

```
// Method call method - not so fast, buster
function square(n){ return n*n };
var i=10000, sum = 0;
while(i--) { sum += square(i) };
```

Into this:

```
// Inline functionality - fasterrrrr!
var i=10000, sum = 0;
while(i--) { sum += i*i };
```

Learn more about JavaScript Performance Rocks!

Chapter 6

Smooth Operators

To == or ===, that is the question

Whether 'tis nobler in the engine to suffer the slings and arrows of haphazard choices, or to take arms against a sea of deceleration!

Actually, there are other questions addressed in this chapterlette, but that is the first one we'll cover.

To == or ===, that is the question

There are two equivalence operators in JavaScript: `==` (equality) and `===` (strict equality).

The strict equality operator (3, count 'em, 3: `===`) gives you a slight edge on performance, in most cases. It's a good choice if you need some pep in your expression.

But be aware, it doesn't behave exactly like the regular ol' two-pronged equality operator, even though it usually seems to.

Watch out for these edge cases:

```
undefined === null
=> FALSE
```

Chapter 2

The Key Pause Approach

Forthright behavior returns stunning results!

Yeah, sure, we just told you to lie, but this technique is a fabulous example of what can go right when you are totally upfront.

Users give up on slow-running apps because of many reasons, but a big one is disappointment. They were expecting it now, dammit!

If you communicate with your visitors up front, they can't be disappointed, by definition—because disappointment comes from the chasm between expectations and reality.

That 5 - 8 Seconds Rule

It's still true that users will wait only 5 - 8 seconds for a page load before abandoning it for dead.

It's also true that you can triple the amount of time they will wait, with a few tweaks to your app's interface.

Wait, wait—TRIPLE?

Yes, that's right, you can triple the amount of time people will wait. Your users will give you vastly more leeway if you just let them know what to

Chapter 3

Batch Client-Side Processes

When you really can't do anything else

The word "batch" summons up delicious memories of warm cookies. This isn't about cookies, nor is it delicious, or even warm. But it might save your butt.

We've recommended that you consider batching for long-running server-side processes. Now we're recommending it for long-running client-side processes, too.

To our way of thinking, you shouldn't have long-running client-side processes. When in doubt, outsource it to the back-end.

But if you really can't avoid it, you need to break it up so your users aren't terribly inconvenienced—or, worse yet, think your app has stopped working, or their browser has crashed.

JavaScript is Blocking

Sometimes you just can't help it and have to process lots of data on the client—which can mean visible temporay 'lock-ups' of your web app: short periods of time where the browser doesn't respond.

This is because JavaScript is blocking and the main browser event loop (the one that handles mouse clicks, the browser menu bar, etc) won't do anything until after the JavaScript has been executed.

Thanks!

for checking out our sample pages
if you'd like to learn more (or even buy!), just click here

JavaScript Performance Rocks!
by Amy Hoy & Thomas Fuchs
http://jsrocks.com/