

24 WAYS to impress your friends

HOME

ARCHIVES

AUTHORS

RSS

TWITTER

DAY

24

23

22

21

20

19

18

17

13

12/2011

Your jQuery: Now With 67% Less Suck

ARTICLE

COMMENTS 45

by [Scott Kosman](#)

Fun fact: [more websites are now using jQuery than Flash](#).

jQuery is an amazing tool that's made JavaScript accessible to developers and designers of all levels of experience. However, as Spiderman taught us, "with great power comes great responsibility." The unfortunate downside to jQuery is that while it makes it easy to write JavaScript, it makes it easy to write really really f*cking bad JavaScript. Scripts that slow down page load, unresponsive user interfaces, and spaghetti code knotted so deep that it should come with a bottle of whiskey for the next sucker developer that has to work on it.

16

15

14

This becomes more important for those of us who have yet to move into the magical fairy wonderland where none of our clients or users view our pages in Internet Explorer. The IE JavaScript engine moves at the speed of an advancing glacier compared to more modern browsers, so optimizing our code for performance takes on an even higher level of urgency.

13

Thankfully, there are a few very simple things anyone can add into their jQuery workflow that can clear up a lot of basic problems. When undertaking code reviews, three of the areas where I consistently see the biggest problems are: inefficient selectors; poor event delegation; and clunky DOM manipulation. We'll tackle all three of these and hopefully you'll walk away with some new jQuery [batarangs](#) to toss around in your next project.

12

11

10

Selector optimization

09

08

SELECTOR SPEED: FAST OR SLOW?

07

06

05

04

Saying that the power behind jQuery comes from its ability to select DOM elements and act on them is like saying that Photoshop is a really good tool for selecting pixels on screen and making them change color – it's a bit of a gross oversimplification, but the fact remains that jQuery gives us a ton of ways to choose which element or elements in a page we want to work with. However, a surprising number of web developers are unaware that all selectors are not created equal; in fact, it's incredible just how drastic the performance difference can be between two selectors that, at first glance, appear nearly identical. For instance, consider these two ways of selecting all paragraph tags inside a `<div>` with an ID.

03

```
$("#id p");
```

```
$("#id").find("p");
```

Would it surprise you to learn that the second way can be more than twice as fast as the first? Knowing which selectors outperform others (and why) is a pretty key building block in making sure your code runs well and doesn't frustrate your users waiting for things to happen.

There are many different ways to select elements using jQuery, but the most common ways can be basically broken down into five different methods. In order, roughly, from fastest to slowest, these are:

- `$("#id");`

This is without a doubt the fastest selector jQuery provides because it maps directly to the native `document.getElementById()` JavaScript method. If possible, the selectors listed below should be prefaced with an ID selector in conjunction with jQuery's `.find()` method to limit the scope of the page that has to be searched (as in the `$("#id").find("p")` example shown above).

- `$("p");`, `$("input");`, `$("form");` and so on

Selecting elements by tag name is also fast, since it maps directly to the native `document.getElementsByTagName()` method.

- `$(".class");`

Selecting by class name is a little trickier. While still performing very well in modern browsers, it can cause some pretty significant slowdowns in IE8 and below. Why? IE9 was the first IE version to support the native `document.getElementsByClassName()` JavaScript method. Older browsers have to resort to using much slower DOM-scraping methods that can really impact performance.

- `$("[attribute=value]");`

There is no native JavaScript method for this selector to use, so the only way that jQuery can perform the search is by crawling the entire DOM looking for matches. Modern browsers that support the `querySelectorAll()` method will perform better in certain cases (Opera, especially, runs these searches much faster than any other browser) but, generally speaking, this type of selector is Slowey McSlowersons.

- `$(":hidden");`

Like attribute selectors, there is no native JavaScript method for this one to use. Pseudo-selectors can be painfully slow since the selector has to be run against every element in your search space. Again, modern browsers with `querySelectorAll()` will perform slightly better here, but try to avoid these if at all possible. If you must use one, try to limit the search space to a specific portion of the page: `$("#list").find(":hidden");`

But, hey, proof is in the performance testing, right? It just so happens that said proof is [sitting right here](#). Be sure to notice the class selector numbers beside IE7 and 8 compared to other browsers and then wonder how the people on the IE team at Microsoft manage to sleep at night. Yikes.

CHAINING

Almost all jQuery methods return a jQuery object. This means that when a method is run, its results are returned and you can continue executing more methods on them. Rather than writing out the same selector multiple times over, just making a selection once allows multiple actions to be run on it.

Without chaining

```
$("#object").addClass("active");  
$("#object").css("color", "#f0f");  
$("#object").height(300);
```

With chaining

```
$("#object").addClass("active").css("color", "#f0f").height(300);
```

This has the dual effect of making your code shorter *and* faster. Chained methods will be slightly faster than multiple methods made on a cached selector, and both ways will be *much* faster than multiple methods made on non-cached selectors. Wait... “cached selector”? What is this new devilry?

CACHING

Another easy way to speed up your code that seems to be a mystery to developers is the idea of caching your selectors. Think of how many times you end up writing the same selector over and over again in any project. Every `$(".element")` selector has to search the entire DOM each time, regardless of whether or not that selector had been previously run. Running the selection once and then storing the results in a variable means that the DOM only has to be searched once. Once the results of a selector have been cached, you can do anything with them.

First, run your search (here we’re selecting all of the `` elements inside `<ul id="blocks">`):

```
var blocks = $("#blocks").find("li");
```

Now, you can use the `blocks` variable wherever you want without having to search the DOM every time.

```
$("#hideBlocks").click(function() {  
    blocks.fadeOut();  
});
```

```
});  
$("#showBlocks").click(function() {  
    blocks.fadeIn();  
});
```

My advice? Any selector that gets run more than once should be cached. [This jsperf test](#) shows just how much faster a cached selector runs compared to a non-cached one (and even throws some chaining love in to boot).

Event delegation

Event listeners cost memory. In complex websites and apps it's not uncommon to have a lot of event listeners floating around, and thankfully jQuery provides some really easy methods for handling event listeners efficiently through delegation.

In a bit of an extreme example, imagine a situation where a 10×10 cell table needs to have an event listener on each cell; let's say that clicking on a cell adds or removes a class that defines the cell's background color. A typical way that this might be written (and something I've often seen during code reviews) is like so:

```
$('.table').find('td').click(function() {  
    $(this).toggleClass('active');  
});
```

jQuery 1.7 has provided us with a new event listener method, [.on\(\)](#). It acts as a utility that wraps all of

jQuery's previous event listeners into one convenient method, and the way you write it determines how it behaves. To rewrite the above `.click()` example using `.on()`, we'd simply do the following:

```
$('.table').find('td').on('click',function() {  
    $(this).toggleClass('active');  
});
```

Simple enough, right? Sure, but the problem here is that we're still binding one hundred event listeners to our page, one to each individual table cell. A far better way to do things is to create one event listener on the table itself that listens for events inside it. Since the majority of events bubble up the DOM tree, we can bind a single event listener to one element (in this case, the `<table>`) and wait for events to bubble up from its children. The way to do this using the `.on()` method requires only one change from our code above:

```
$('.table').on('click','td',function() {  
    $(this).toggleClass('active');  
});
```

All we've done is moved the `td` selector to an argument inside the `.on()` method. Providing a selector to `.on()` switches it into delegation mode, and the event is only fired for descendants of the bound element (`table`) that match the selector (`td`). With that one simple change, we've gone from having to bind one hundred event listeners to just one. You might think that the browser having to do one hundred times less work would be a good thing and you'd be completely right. The [difference between the two examples above](#) is staggering.

(Note that if your site is using a version of jQuery earlier than 1.7, you can accomplish the very same thing using the `.delegate()` method. The syntax of how you write the function differs slightly; if you've never used it before, it's worth checking the API docs for that page to see how it works.)

DOM manipulation

jQuery makes it very easy to manipulate the DOM. It's trivial to create new nodes, insert them, remove other ones, move things around, and so on. While the code to do this is simple to write, every time the DOM is manipulated, the browser has to repaint and reflow content which can be extremely costly. This is no more evident than in a long loop, whether it be a standard `for()` loop, `while()` loop, or `jQuery $.each()` loop.

In this case, let's say we've just received an array full of image URLs from a database or Ajax call or wherever, and we want to put all of those images in an unordered list. Commonly, you'll see code like this to pull this off:

```
var arr = [reallyLongArrayOfImageURLs];
$.each(arr, function(count, item) {
    var newImg = '<li></li>';
    $('#imgList').append(newImg);
});
```

There are a couple of problems with this. For one (which you should have already noticed if you've read the earlier part of this article), we're making the `$("#imgList")` selection once for each iteration of our loop. The other problem here is that each time the loop iterates, it's adding a new

`` to the DOM. Each of those insertions is going to be costly, and if our array is quite large then this could lead to a massive slowdown or even the dreaded 'A script is causing this page to run slowly' warning.

```
var arr = [reallyLongArrayOfImageURLs],
    tmp = '';
$.each(arr, function(count, item) {
    tmp += '<li></li>';
});
$('#imgList').append(tmp);
```

All we've done here is create a `tmp` variable that each `` is added to as it's created. Once our loop has finished iterating, that `tmp` variable will contain all of our list items in memory, and can be appended to our `` all in one go. Browsers work much faster when working with objects in memory rather than on screen, so this is a much faster, more CPU-cycle-friendly method of building a list.

Wrapping up

These are far from being the only ways to make your jQuery code run better, but they are among the simplest ones to implement. Though each individual change may only make a few milliseconds of difference, it doesn't take long for those milliseconds to add up. Studies have shown that the human eye can discern delays of as few as 100ms, so simply making a few changes sprinkled throughout your code can very easily have a noticeable effect on how well your website or app performs. Do you have other jQuery optimization tips to share? Leave them in the comments and help make us all

better.

Now go forth and make awesome!

Like what you read? [Tweet this article](#) or [Leave a comment](#)

About the author

A 35 year old Canadian expat currently plying his trade as Associate Director of Standards Architecture at Crispin Porter + Bogusky in Göteborg, Sweden, **Scott Kosman** can also be found on the [Tweeters](#) and some of his other work can even be found on the [internet](#). He likes JavaScript, bicycles, cats, bacon, and thinks you're pretty awesome.



[More information](#)

Related articles

Using Google App Engine as Your Own Content
Delivery Network

06/12/2008 by [Matt Riggott](#)

Performance On A Shoe String

24/12/2007 by [Drew McLellan](#)

Minification: A Christmas Diet

06/12/2007 by [Gareth Rushgrove](#)

[Article archives...](#)

In association with:

Perch
a really little cms



The easiest way to publish **HTML5** websites your clients will love.

24 WAYS IS AN [EDGE OF MY SEAT.COM](#) PRODUCTION. EDITED BY [DREW MCLELLAN](#) AND [BRIAN SUDA](#).
ASSISTED BY [ANNA DEBENHAM](#) AND [OWEN GREGORY](#). DESIGN DELIVERED BY [MADE BY ELEPHANT](#).
POSSIBLE ONLY WITH THE HELP OF [OUR DAZZLING AUTHORS](#). GRAB OUR [RSS FEED](#).
FOLLOW US ON [TWITTER](#), [FACEBOOK](#) OR [GOOGLE+](#) FOR DAILY UPDATES.