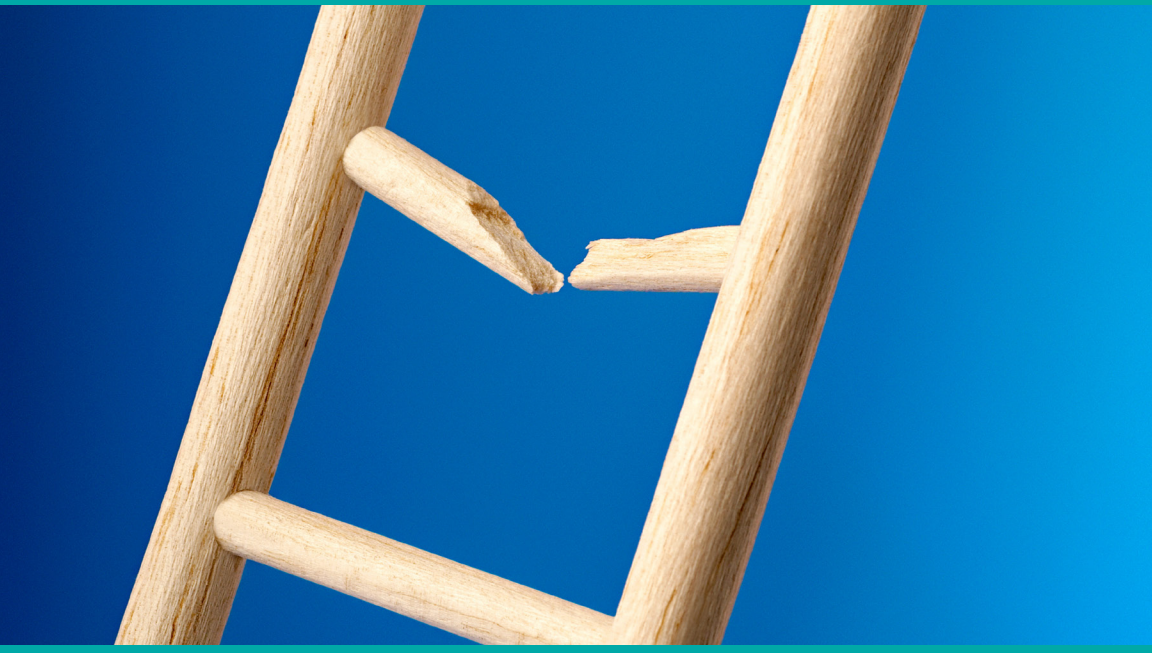
The Problem with Native JavaScript APIs

External JavaScript Libraries Still Matter



Nicholas C. Zakas

O'REILLY®

The Problem with Native JavaScript APIs

Nicholas C. Zakas

O'REILLY®
Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

The Problem with Native JavaScript APIs
by Nicholas C. Zakas

Published by O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

O'Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://my.safaribooksonline.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Editor: Mac Slocum
Production Editor: Melanie Yarbrough
Cover Designer: Karen Montgomery
Interior Designer: David Futato
Illustrator: Robert Romano

July 2012: First Edition.

Revision History for the First Edition:
2012-07-20 First release
See *http://oreilly.com/catalog/errata.csp?isbn=9781449339951* for release details.

ISBN: 978-1-449-33995-1
1344364955

Table of Contents

The Problem with Native JavaScript APIs

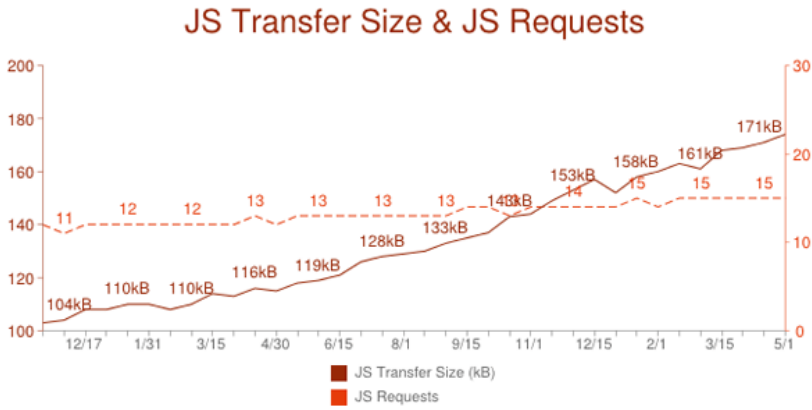
The past couple of years have seen unprecedented changes in web browser technology. For most of the history of the Web, change came at an agonizingly slow pace, as minor features took years to stabilize and roll out across browsers. Then came HTML5. All of a sudden, browsers were being released faster. New features were being implemented almost as soon as they had been spec'd. Features inspired by popular JavaScript libraries became standardized as part of the browser's JavaScript API.

Shortly after that, there came a wave of proclamations. "You don't even need a JavaScript library anymore," some said. "You can just use the native JavaScript APIs to do the same thing." Looking at the browser's JavaScript landscape, it's easy to see why people would say that. Retrieving document object model (DOM) elements using CSS selectors is natively supported through `querySelector()` and `querySelectorAll()`. This capability was heavily inspired by jQuery, the library responsible for popularizing the use of CSS selectors to retrieve and manipulate DOM elements. You can also retrieve elements simply using a CSS class via the `getElementsByClassName()` method, which is based on the method of the same name in the Prototype JavaScript library. Add to those features native drag-and-drop, cross-domain Ajax, cross-iframe communication, client-side data storage, form validation, and a whole host of others, and it seems like browsers now are doing natively what you previously always needed a JavaScript library to do. Does that mean it's time to give up our libraries in favor of native APIs?

The answer to that question is not only "no," but I would take it one step further to say that you shouldn't be using native JavaScript APIs directly at all. To be clear, I'm not talking about ECMAScript APIs, which represent the core functionality of JavaScript. I'm talking about those APIs related to the browser object model (BOM) and the DOM.

Issues with Native APIs

Don't get me wrong, I'm very glad to have all of these new capabilities in browsers. Web browsers are becoming more powerful by the day, allowing for better user experiences, including those that can match or exceed desktop experiences. With all of this advancement, the amount of JavaScript that the average developer writes is growing all the time. In the past year and a half, the average number of JavaScript requests per page has increased from 11 to 15, and the average amount of JavaScript (compressed) has increased from 104kb to 171kb (source: HTTP Archive).



JavaScript Transfer Size & JavaScript Requests - Alexa Top 1000.

Given this drastic increase in the amount of JavaScript per page, coupled with the still growing number of browsers to support, web developers have a tougher job than ever before. We are all writing far more JavaScript code now than we were even five years ago. The more code you write, the harder maintaining it becomes. Relying on native JavaScript APIs to get the job done puts you at a severe disadvantage.

Browsers are written by humans just like web pages are written by humans. All humans have one thing in common: They make mistakes. Browsers have bugs just like web pages have bugs just like any other software has bugs. The native APIs you're relying on likely have bugs. And these bugs don't necessarily even mean the browser developer did something wrong; it could have been due to a misunderstanding or misinterpretation of the specification. These sorts of things happen all the time. Internet Explorer was famous for all manner of JavaScript API bugs — for example, `getElementById()` also returning elements whose `name` attribute matched a given ID. These subtle browser differences lead to bugs.

When you use native JavaScript APIs directly, you are placing a bet. That bet is that all browsers implement the API exactly the same. You're banking your future development time on it. And if a browser implements that API incorrectly, what is your course of action? How quickly can you roll out a fix to your users? You'll start writing workarounds and browser detection, and all of a sudden your code isn't as straightforward to maintain. And sometimes the differences in the browsers are so great that a simple workaround won't do.

Case Study: matchMedia()

The `matchMedia()` method is defined in the CSS Object Model (CSSOM) Views specification as a way to manage CSS media queries in JavaScript. The method takes a single argument, CSS media query, and returns an object that represents that query. That object can be used to determine if the query currently matches and to assign a listener that is fired when the query begins to match and stops matching. Here's some example code:

```
var result = window.matchMedia("(orientation:portrait)");

result.addListener(function(match){

    if (match.media == "(orientation:portrait)") {

        if (match.matches) {

            console.log("It matches now!");

        } else {

            console.log("It doesn't match anymore!");

        }

    }

});
```

This code monitors the browser to see when it's being used in portrait mode. The listener will fire both when the browser is put into portrait mode and when it comes out of portrait mode. This is very handy to alter your JavaScript behavior based on what the browser window is doing.

You can use this method in Internet Explorer 10+ (as `msMatchMedia()`), Chrome, Safari 5.1+, Firefox 9+, and Safari for iOS 5+. However, until very recently, there were some bugs that caused this functionality to work very differently across browsers.

In Firefox, there was a bug when using `matchMedia()` in this way:

```
window.matchMedia("screen and (max-width:600px)").addListener
 (function(media){

    console.log("HERE: " + window.innerWidth);

});
```

This pattern doesn't keep the results of `matchMedia()` in a variable, instead immediately attaching a listener. Firefox would effectively lose the listener when this pattern was used. Sometimes the listener would fire, but most of the time it would never fire, regardless of window resizing. The only workaround was to ensure that the result of `matchMedia()` was being stored in a variable so the reference was never destroyed. This bug has since been fixed in Firefox.

Around the same time, WebKit also had a bug in its implementation. This bug was a little more tricky. Basically, the first call to `matchMedia()` would return the correct value for `matches`, but that value would never be updated, and listeners would never be fired unless there was CSS on the page that made use of the same media query. So, in order for the code in the previous example to work, you would need to have something like this in your CSS:

```
@media screen and (max-width:600px) {

    .foo {}

}
```

The media query block in your CSS needed to have at least one rule, although the rule could be empty. This was a huge amount of overhead to web developers in order to get this feature to work. This bug has also since been fixed.

The `matchMedia()` method is a perfect example of how native APIs in different browsers can be developed according to one specification and still end up with different issues. Most such issues are subtle and difficult to track down. In this particular case, if you are using the native `matchMedia()` method, you would likely have needed to pull it out and replace it with a facade that could add the extra code in for you.

This case study also underscores an important point: If you do find yourself using native APIs, help out the entire web development community by filing bugs when you find incompatibilities and issues. Browser vendors are now working faster than ever to fix compatibility issues. Both of the issues I found with `matchMedia()` caused me to file bugs with Firefox and WebKit. Not sure how to file bugs for a browser? Read John Resig's excellent post on the subject.

Facades and Polyfills

A facade is a design pattern that creates a different interface for a feature. The goal of a facade is to abstract away some underlying interface so that you don't need to access it directly. All of your interaction goes through the facade, which allows you to manipulate the operation of the underlying functionality as necessary. There's nothing magical about a facade. If you've ever used jQuery or YUI, then you've used facades.

Polyfills (aka shims) are a bit different. A polyfill tries to implement a native API directly. Paul Irish attempted to create a polyfill for `matchMedia()`, defining his own function with that name if one didn't already exist. There are other polyfills that you're probably familiar with, such as Modernizr, that seek to fill in other pieces of missing functionality in the browser.

When there's a choice between facades and polyfills, I always choose the facade. The reason is that polyfills suffer from the same downsides as native APIs. They represent yet another implementation of the same functionality. The big problem for polyfills is determining which implementation to follow. Taking the `matchMedia()` method as an example: which of the two strange bugs will the polyfill mimic? Again, the goal is to have your application logic completely free of browser-specific code. Accessing `matchMedia()` directly from your application logic assumes that it works the same everywhere, but in fact, it would behave in at least three different ways: the Firefox way, the WebKit way, and the polyfill way. Polyfills just don't give you enough protection from underlying browser differences.

On the other hand, using a facade allows you to completely abstract away the browser differences from your application code. The facade doesn't even need the same method signature or objects; it just needs to provide the same functionality. I wrote a `matchMedia()` facade as a YUI module so that I could use it in my application. It's quite different from the native version, which allowed me to work around the various browser bugs to get a consistent experience. Here's how it would look in your application code:

```
        YUI({

        //Last Gallery Build of this module

        gallery: 'gallery-2012.01.18-21-09'

    }).use('gallery-media', function(Y) {


        //detect current media state

        console.log(Y.Media.matches("screen and (max-width:600px)"));
```

```
        //subscribe to change in media state

        Y.Media.on("screen and (max-width:600px)", function(result) {

            console.log(result.media + " now " +
            (result.matches ? "matches" : "doesn't match"));

        });


    });
```

In some cases, a facade may end up implementing a feature that doesn't exist (as I did in this example), which makes it seem more like a polyfill. The difference is that a polyfill implements an already-existing interface while a facade is implementing the functionality without the interface. The latter is much easier to deal with in the long term since there aren't synchronization issues.

Facades give you a big advantage in keeping your code maintainable and ensuring your application logic doesn't need to know which browser is being used.

What to Do?

In my talk, "Scalable JavaScript Application Architecture" (video, slides), I made the point that your application should never need to know which browser it's running in. Application logic should be written one way for all browsers in order to keep the code maintainable. If you're using native APIs in your application logic, you can't help but know what browser is being used because you need to account for browser differences. That means your application logic will always need to be updated as new browsers and new browser versions are released. That's a recipe for disaster.

You should absolutely be using a JavaScript library to abstract away browser differences for you. That is the appropriate location in your JavaScript architecture for browser-specific code to exist. Libraries like jQuery, YUI, and Dojo abstract away browser differences behind facades, which allow you to focus on building your application logic in a browser-agnostic way. This gives you a great advantage over using native APIs directly: Browser incompatibilities can be changed within the library and the rest of your application logic doesn't have to change at all. What's more, these popular JavaScript libraries are actively maintained and tested against new browsers, so upgrading is all you have to do to get the latest fixes. In a world where JavaScript's applications are

getting more and more complex, using a JavaScript library gives you a big advantage as your application continues to grow and evolve.

So, keep using your favorite JavaScript library. Don't be tempted by the draw of native APIs simply because you can avoid downloading an external library. Using native APIs comes with a high cost of maintainability down the road.

Thanks to Paul Irish, Axel Rauschmayer, and Marco Rogers for reviewing an early draft of this article.

About the Author

Nicholas C. Zakas is a front-end consultant, author, and speaker. He worked at Yahoo! for almost five years, where he was front-end tech lead for the Yahoo! homepage and a contributor to the YUI library. He is the author of *Professional JavaScript for Web Developers* (Wrox, 2012), *Professional Ajax* (Wrox, 2007), and *High Performance JavaScript* (O'Reilly, 2010). Nicholas is a strong advocate for development best practices including progressive enhancement, accessibility, performance, scalability, and maintainability. He blogs regularly at *www.nczonline.net* and can be found on Twitter via @slicknet.