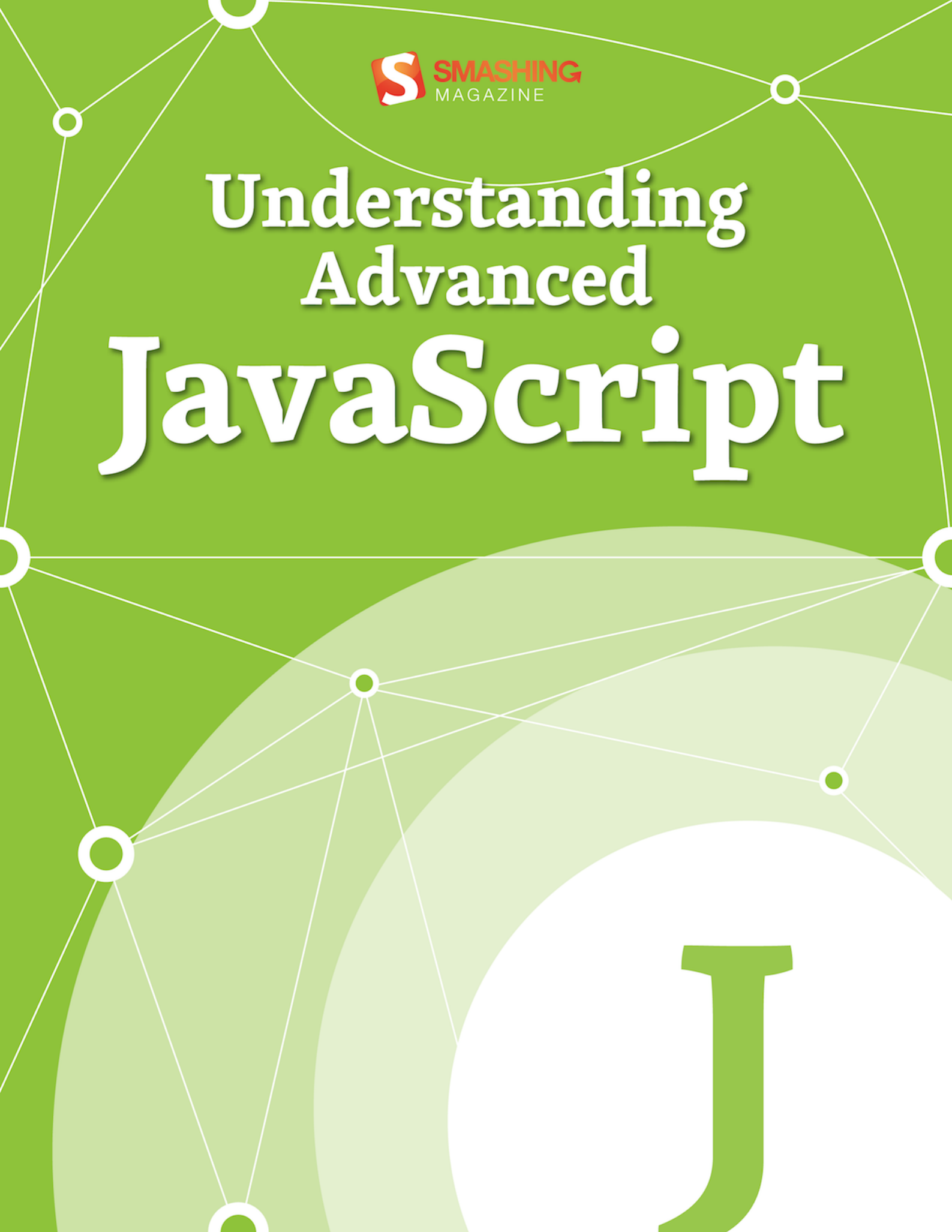
SMASHING MAGAZINE

Understanding Advanced JavaScript



Imprint

© 2013 Smashing Media GmbH, Freiburg, Germany
ISBN: 978-3-943075-60-1 (Version 1: February 2013)
Cover Design: Ricardo Gimenes
PR & Press: Stephan Poppe
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.
Copywriter: Lucy Leiderman.
Idea & Concept: Smashing Media GmbH

About This Book

This book is an exploration of popular advanced JavaScript concepts for those who already have a grasp on the basics. *Understanding Advanced JavaScript* is a comprehensive manual and how-to guide about all things JavaScript. Learn to design better APIs, use the latest tools and navigate the JavaScript MVC Jungle effortlessly. In this eBook, you will receive expert tips and techniques on avoiding coding tripwires and improving your programming methodologies. Need to know how your JavaScript performs? Find out about unit testing and user- and network-related analyses. Master JavaScript with industry leaders who explain how to make fine-tuning, streamlining and testing your JavaScript code easy.

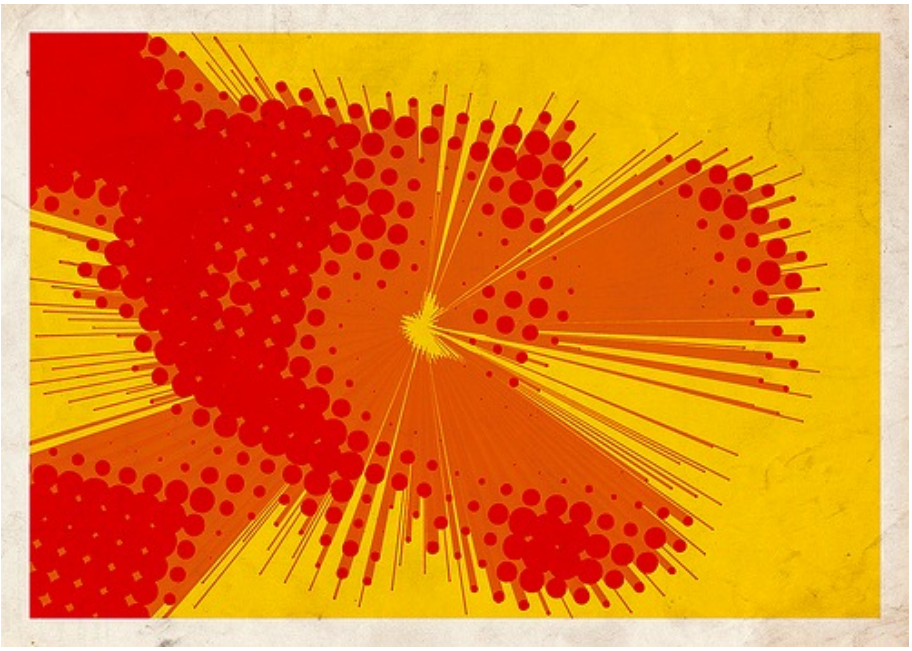
TABLE OF CONTENTS

Analyzing Network Characteristics Using JavaScript And The DOM 3
Introduction To JavaScript Unit Testing 13
Journey Through The JavaScript MVC Jungle 34
JavaScript Events And Responding To The User 58
JavaScript Profiling With The Chrome Developer Tools 74
Writing Fast, Memory-Efficient JavaScript 87
Designing Better JavaScript APIs 118
About The Authors 153

Analyzing Network Characteristics Using JavaScript And The DOM

BY PHILIP TELLIS

As Web developers, we have an affinity for developing with JavaScript. Whatever the language used in the back end, JavaScript and the browser are the primary language-platform combination available at the user's end. It has many uses, ranging from silly to experience-enhancing.



(Image: Viktor Hertz[1]*)*

In this chapter, we'll look at some methods of manipulating JavaScript to determine various network characteristics from within the browser—characteristics that were previously available only to applications that directly interface with the operating system. Much of this was discovered while building the Boomerang[2] project to measure real user performance.

1. http://www.flickr.com/photos/hertzen/4878410201/in/photostream
2. http://github.com/bluesmoon/boomerang

What's In A Network Anyway?

The network has many layers, but the Web developers among us care most about HTTP, which runs over TCP[3] and IP[4] (otherwise known jointly as the Internet protocol suite[5]). Several layers are below that, but for the most part, whether it runs on copper, fiber or homing[6] pigeons[7] does not affect the layers or the characteristics that we care about.

NETWORK LATENCY

Network latency is typically the time it takes to send a signal across the network and get a response. It's also often called roundtrip time or ping time because it's the time reported by the `PING` command. While this is interesting to network engineers who are diagnosing network problems, Web developers care more about the time it takes to make an HTTP request and get a response. Therefore, we'll define HTTP latency as the time it takes to make the smallest HTTP request possible, and to get a response with insignificant server-processing time (i.e. the only thing the server does is send a response).

Cool tip: Light and electricity travel through fiber[8] and copper at 66% the speed of light in a vacuum, or 20 × 100.000.000 kilometres per second. A good approximation of network latency between points A and B is four times[9] the time it takes light or electricity to travel the distance. Greg's Cable Map[10] is a good resource to find out the length and bandwidth of undersea network cables. I'll leave it to you to put these pieces together.

NETWORK THROUGHPUT

Network throughput tells us how well a network is being utilized. We may have a 3-megabit network connection but are effectively using only 2 megabits because the network has a lot of idle time.

DNS

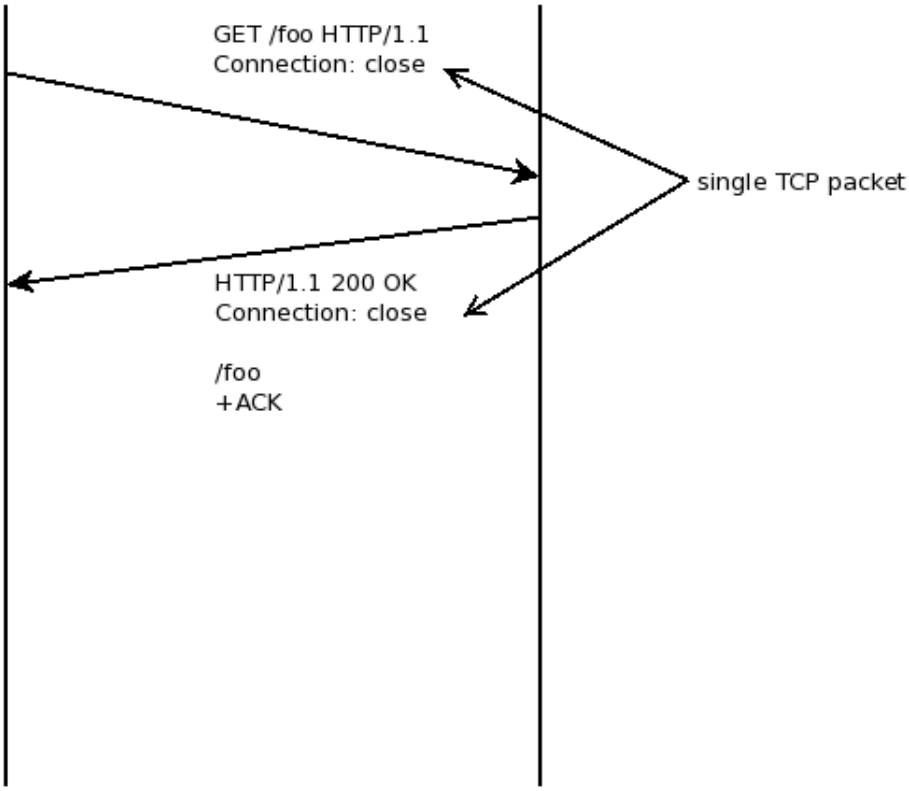
DNS is a little different from everything else we care about. It works over UDP and typically happens at a layer that is transparent to

3. http://en.wikipedia.org/wiki/Transmission_Control_Protocol
4. http://en.wikipedia.org/wiki/Internet_Protocol
5. http://en.wikipedia.org/wiki/Internet_protocol_suite
6. http://www.faqs.org/rfcs/rfc2549.html
7. http://en.wikipedia.org/wiki/IP_over_Avian_Carriers
8. http://en.wikipedia.org/wiki/Optical_fiber
9. http://rescomp.stanford.edu/~cheshire/rants/Latency.html
10. http://cablemap.info/

JavaScript. We'll see how best to ascertain the time it takes to do a DNS lookup.

There is, of course, much more to the network, but determining these characteristics through JavaScript in the browser gets increasingly harder.

Measuring Network Latency With JavaScript



My first instinct was that measuring latency simply entailed sending one packet each way and timing it. It's fairly easy to do this in JavaScript:

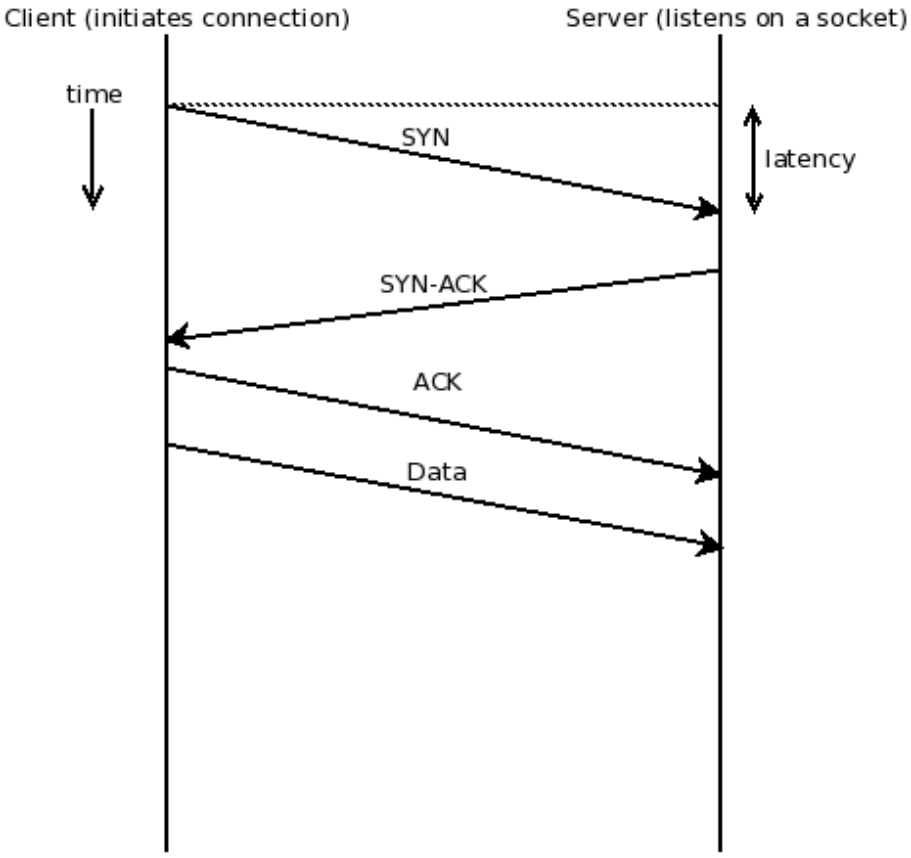
```
var ts, rtt, img = new Image;
img.onload=function() { rtt=(+new Date - ts) };
ts = +new Date;
img.src="/1x1.gif";
```

We start a timer, then load a 1 × 1 pixel GIF and measure when its `onload` event fires. The GIF itself is 35 bytes in size and so fits in a single TCP packet even with HTTP headers added in.

This kinda sorta works, but has inconsistent results. In particular, the first time you load an image, it will take a little longer than subsequent loads—even if we make sure the image isn't cached. Looking at

the TCP packets that go across the network explains what's happening, as we'll see in the following section.

TCP HANDSHAKE AND HTTP KEEP-ALIVE



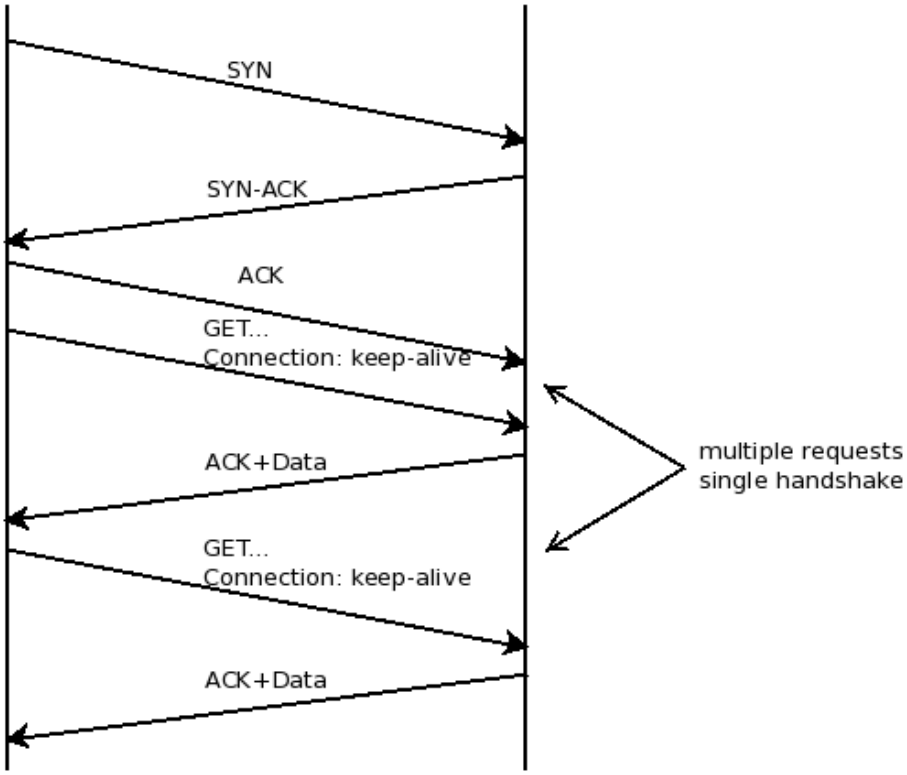
When loading a Web page or image or any other Web resource, a browser opens a TCP connection to the specified Web server, and then makes an HTTP `GET` request over this connection. The details of the TCP connection and HTTP request are hidden from users and from Web developers as well. They are important, though, if we need to analyze the network's characteristics.

The first time a TCP connection is opened between two hosts (the browser and the server, in our case), they need to "handshake." This takes place by sending three packets between the two hosts. The host that initiates the connection (the browser in our case) first sends a SYN packet, which kind of means, "Let's SYNc up. I'd like to talk to you. Are you ready to talk to me?" If the other host (the server in our case) is ready, it responds with an ACK, which means, "I ACKnowledge your SYN." And it also sends a SYN of its own, which means, "I'd like to SYNc up, too. Are you ready?" The Web browser then completes the handshake with its own ACK, and the connection is established. The connection could fail, but the process behind a connection failure is beyond the scope of this chapter.

Once the connection is established, it remains open until both ends decide to close it, by going through a similar handshake.

When we throw HTTP over TCP, we now have an HTTP client (typically a browser) that initiates the TCP connection and sends the first data packet (a `GET` request, for example). If we're using HTTP/1.1 (which almost everyone does today), then the default will be to use HTTP keep-alive (`Connection: keep-alive`). This means that several HTTP requests may take place over the same TCP connection. This is good, because it means that we reduce the overhead of the handshake (three extra packets).

Now, unless we have HTTP pipelining[11] turned on (and most browsers and servers turn it off), these requests will happen serially.



We can now modify our code a bit to take the time of the TCP handshake into account, and measure latency accordingly.

```
var t=[], n=2, tcp, rtt;
var ld = function() {
   t.push(+new Date);
   if(t.length > n)
     done();
   else {
```

11. http://en.wikipedia.org/wiki/HTTP_pipelining

```
      var img = new Image;
      img.onload = ld;
      img.src="/1x1.gif?" + Math.random()
                          + '=' + new Date;
   }
};
var done = function() {
  rtt=t[2]-t[1];
  tcp=t[1]-t[0]-rtt;
};
ld();
```

With this code, we can measure both latency and the TCP handshake time. There is a chance that a TCP connection was already active and that the first request went through on that connection. In this case, the two times will be very close to each other. In all other cases, `rtt`, which requires two packets, should be approximately 66% of `tcp`, which requires three packets. Note that I say "approximately," because network jitter and different routes at the IP layer can make two packets in the same TCP connection take different lengths of time to get through.

You'll notice here that we've ignored the fact that the first image might have also required a DNS lookup. We'll look at that in part 2.

Measuring Network Throughput With JavaScript

Again, our first instinct with this test was just to download a large image and measure how long it takes. Then `size/time` should tell us the throughput.

For the purpose of this code, let's assume we have a global object called `image`, with details of the image's URL and size in bits.

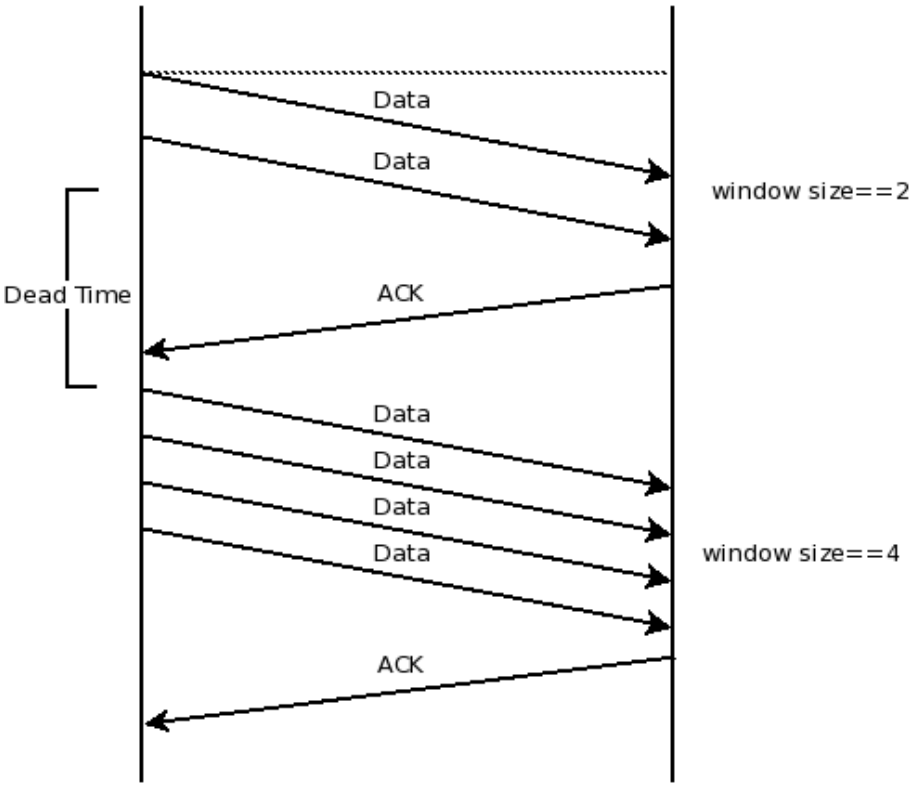
```
// Assume global object
// image={ url: …, size: … }
var ts, rtt, bw, img = new Image;
img.onload=function() {
   rtt=(+new Date - ts);
   bw = image.size*1000/rtt;    // rtt is in ms
};
ts = +new Date;
img.src=image.url;
```

Once this code has completed executing, we should have the network throughput in kilobits per second stored in `bw`.

Unfortunately, it isn't that simple, because of something called TCP slow-start[12].

SLOW-START

In order to avoid network congestion, both ends of a TCP connection will start sending data slowly and wait for an acknowledgement (an ACK packet). Remember than an ACK packet means, "I ACKnowledge what you just sent me." Every time it receives an ACK without timing out, it assumes that the other end can operate faster and will send out more packets before waiting for the next ACK. If an ACK doesn't come through in the expected timeframe, it assumes that the other end cannot operate fast enough and so backs off.



This means that our throughput test above would have been fine as long as our image is small enough to fit within the current TCP window, which at the start is set to 2. While this is fine for slow networks, a fast network really wouldn't be taxed by so small an image.

Instead, we'll try by sending across images of increasing size and measuring the time each takes to download.

12. http://en.wikipedia.org/wiki/Slow-start

For the purpose of the code, the global `image` object is now an array with the following structure:

```
var image = [
  {url: …, size: … }
];
```

An array makes it easy to iterate over the list of images, and we can easily add large images to the end of the array to test faster network connections.

```
var i=0;
var ld = function() {
   if(i > 0)
      image[i-1].end = +new Date;
   if(i >= image.length)
      done();
   else {
      var img = new Image;
      img.onload = ld;
      image[i].start = +new Date;
      img.src=image[i].url;
   }
   i++;
};
```

Unfortunately, this breaks down when a very slow connection hits one of the bigger images; so, instead, we add a `timeout` value for each image, designed so that we hit upon common network connection speeds quickly. Details of the image sizes and `timeout` values are listed in this spreadsheet[13].

Our code now looks like this:

```
var i=0;
var ld = function() {
   if(i > 0) {
      image[i-1].end = +new Date;
      clearTimeout(image[i-1].timer);
   }
```

13. https://spreadsheets.google.com/ccc?key=0AplxPyCzmQi6dDRBN2JEd190N1hhV1N5cHQtUVdBMUE&hl=en_GB

```
        if(i >= image.length ||
              (i > 0 && image[i-1].expired))
           done();
        else {
           var img = new Image;
           img.onload = ld;
           image[i].start = +new Date;
           image[i].timer =
                 setTimeout(function() {
                           image[i].expired=true
                        },
                        image[i].timeout);
           img.src=image[i].url;
        }
        i++;
     };
```

This looks much better—and works much better, too. But we'd see much variance between multiple runs. The only way to reduce the error in measurement is to run the test multiple times and take a summary value, such as the median. It's a tradeoff between how accurate you need to be and how long you want the user to wait before the test completes. Getting network throughput to an order of magnitude is often as close as you need to be. Knowing whether the user's connection is around 64 Kbps or 2 Mbps is useful, but determining whether it's exactly 2048 or 2500 Kbps is much less useful.

Summary And References

We've looked at how the packets that make up a Web request get through between browser and server, how this changes over time, and how we can use JavaScript and a little knowledge of statistics to make educated guesses at the characteristics of the network that we're working with.

Here's a list of links to resources that were helpful in compiling this chapter.

- Greg's Cable Map[14], Greg Mahlknecht
  Aggregated information about undersea network cables.

14. http://cablemap.info/

- "Transmission Control Protocol[15]," Wikipedia
  This page links to all of the RFCs.
- *TCP/IP Illustrated, Vol. 1: The Protocols*[16], by Richard Stevens
  A link to the book on Amazon.
- "Slow-Start[17]," Wikipedia
  Also, see the links above for congestion control and Nagle's Algorithm.
- "Bandwidth Images Sizes[18]," Google Spreadsheet
  This is based on the research done while building Boomerang.
- Boomerang[19]
  The Boomerang project on GitHub, where much of this has been implemented.

15. http://en.wikipedia.org/wiki/Transmission_Control_Protocol
16. http://www.amazon.com/TCP-Illustrated-Vol-Addison-Wesley-Professional/dp/0201633469
17. http://en.wikipedia.org/wiki/Slow-start
18. https://spreadsheets.google.com/ccc?key=0AplxPyCzmQi6dDRBN2JEd190N1hhV1N5cHQtUVdBMUE&hl=en_GB
19. http://github.com/bluesmoon/boomerang/

Introduction To JavaScript Unit Testing

BY JÖRN ZAEFFERER

You probably know that testing is good, but the first hurdle to overcome when trying to write unit tests for client-side code is the lack of any actual units; JavaScript code is written for each page of a website or each module of an application and is closely intermixed with back-end logic and related HTML. In the worst case, the code is completely mixed with HTML, as inline events handlers.

This is likely the case when no JavaScript library for some DOM abstraction is being used; writing inline event handlers is much easier than using the DOM APIs to bind those events. More and more developers are picking up a library such as jQuery to handle the DOM abstraction, allowing them to move those inline events to distinct scripts, either on the same page or even in a separate JavaScript file. However, putting the code into separate files doesn't mean that it is ready to be tested as a unit.

What is a unit anyway? In the best case, it is a pure function that you can deal with in some way—a function that always gives you the same result for a given input. This makes unit testing pretty easy, but most of the time you need to deal with side effects, which here means DOM manipulations. It's still useful to figure out which units we can structure our code into and to build unit tests accordingly.

Building Unit Tests

With that in mind, we can obviously say that starting with unit testing is much easier when starting something from scratch. But that's not what this chapter is about. This chapter is to help you with the harder problem: extracting existing code and testing the important parts, potentially uncovering and fixing bugs in the code.

The process of extracting code and putting it into a different form, without modifying its current behavior, is called refactoring. Refactoring is an excellent method of improving the code design of a program; and because any change could actually modify the behaviour of the program, it is safest to do when unit tests are in place.

This chicken-and-egg problem means that to add tests to existing code, you have to take the risk of breaking things. So, until you have solid coverage with unit tests, you need to continue manually testing to minimize that risk.

That should be enough theory for now. Let's look at a practical example, testing some JavaScript code that is currently mixed in with and connected to a page. The code looks for links with `title` attributes, using those titles to display when something was posted, as a relative time value, like "5 days ago":

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
<title>Mangled date examples</title>
<script>
function prettyDate(time){
    var date = new Date(time || ""),
    diff = ((new Date().getTime() - date.getTime()) /
1000),
    day_diff = Math.floor(diff / 86400);

        if (isNaN(day_diff) || day_diff < 0 || day_diff
>= 31) {
                return;
        }

        return day_diff == 0 && (
                        diff < 60 && "just now" ||
                        diff < 120 && "1 minute ago" ||
                        diff < 3600 && Math.floor( diff
/ 60 ) + " minutes ago" ||
                        diff < 7200 && "1 hour ago" ||
                        diff < 86400 && Math.floor( diff
/ 3600 ) + " hours ago") ||
                day_diff == 1 && "Yesterday" ||
                day_diff < 7 && day_diff + " days ago" ||
                day_diff < 31 && Math.ceil( day_diff / 7
) + " weeks ago";
}

window.onload = function(){
        var links = document.getElementsByTagName("a");
        for (var i = 0; i < links.length; i++) {
                if (links[i].title) {
```

```
            var date = prettyDate(links[i].title);
            if (date) {
                    links[i].innerHTML = date;
            }
          }
        }
};
</script>
</head>
<body>

<ul>
<li class="entry" id="post57">
        <p>blah blah blah…</p>
        <small class="extra">
                Posted <a href="/2008/01/blah/57/"
title="2008-01-28T20:24:17Z">January 28th, 2008</a>
                by <a href="/john/">John Resig</a>
        </small>
</li>
<!-- more list items -->
</ul>

</body>
</html>
```

If you ran that example, you'd see a problem: none of the dates get replaced. The code works, though. It loops through all anchors on the page and checks for a `title` property on each. If there is one, it passes it to the `prettyDate` function. If `prettyDate` returns a result, it updates the `innerHTML` of the link with the result.

Make Things Testable

The problem is that for any date older then 31 days, `prettyDate` just returns undefined (implicitly, with a single `return` statement), leaving the text of the anchor as is. So, to see what's supposed to happen, we can hardcode a "current" date:

```
<!DOCTYPE html>
<html>
<head>
```

```
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
<title>Mangled date examples</title>
<script>
function prettyDate(now, time){
        var date = new Date(time || ""),
                diff = (((new Date(now)).getTime() -
date.getTime()) / 1000),
                day_diff = Math.floor(diff / 86400);

        if (isNaN(day_diff) || day_diff < 0 || day_diff
>= 31) {
                return;
        }

        return day_diff == 0 && (
                        diff < 60 && "just now" ||
                        diff < 120 && "1 minute ago" ||
                        diff < 3600 && Math.floor( diff
/ 60 ) + " minutes ago" ||
                        diff < 7200 && "1 hour ago" ||
                        diff < 86400 && Math.floor( diff
/ 3600 ) + " hours ago") ||
                day_diff == 1 && "Yesterday" ||
                day_diff < 7 && day_diff + " days ago" ||
                day_diff < 31 && Math.ceil( day_diff / 7
) + " weeks ago";
}

window.onload = function(){
        var links = document.getElementsByTagName("a");
        for (var i = 0; i < links.length; i++) {
                if (links[i].title) {
                        var date =
prettyDate("2008-01-28T22:25:00Z", links[i].title);
                        if (date) {
                            links[i].innerHTML = date;
                        }
                }
        }
};
```

```
</script>
</head>
<body>

<ul>
<li class="entry" id="post57">
        <p>blah blah blah…</p>
        <small class="extra">
                Posted <a href="/2008/01/blah/57/"
title="2008-01-28T20:24:17Z">January 28th, 2008</a>
                by <a href="/john/">John Resig</a>
        </small>
</li>
<!-- more list items -->
</ul>

</body>
</html>
```

- Run this example.[20]

Now, the links should say "2 hours ago," "Yesterday" and so on. That's something, but still not an actual testable unit. So, without changing the code further, all we can do is try to test the resulting DOM changes. Even if that did work, any small change to the markup would likely break the test, resulting in a really bad cost-benefit ratio for a test like that.

Refactoring, Stage 0

Instead, let's refactor the code just enough to have something that we can unit test.

We need to make two changes for this to happen: pass the current date to the `prettyDate` function as an argument, instead of having it just use `new Date`, and extract the function to a separate file so that we can include the code on a separate page for unit tests.

20. http://provide.smashingmagazine.com/introduction-to-js-unit-testing-code/1-mangled.html

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
<title>Refactored date examples</title>
<script src="prettydate.js"></script>
<script>
window.onload = function() {
        var links = document.getElementsByTagName("a");
        for ( var i = 0; i < links.length; i++ ) {
                if (links[i].title) {
                        var date =
prettyDate("2008-01-28T22:25:00Z", links[i].title);
                        if (date) {
                            links[i].innerHTML = date;
                        }
                }
        }
};
</script>
</head>
<body>

<ul>
<li class="entry" id="post57">
        <p>blah blah blah…</p>
        <small class="extra">
                Posted <a href="/2008/01/blah/57/"
title="2008-01-28T20:24:17Z">January 28th, 2008</a>
                by <a href="/john/">John Resig</a>
        </small>
</li>
<!-- more list items -->
</ul>

</body>
</html>
```

Here's the contents of `prettydate.js`:

```
function prettyDate(now, time){
    var date = new Date(time || ""),
        diff = (((new Date(now)).getTime() -
date.getTime()) / 1000),
        day_diff = Math.floor(diff / 86400);

    if (isNaN(day_diff) || day_diff < 0 || day_diff >=
31) {
        return;
    }

    return day_diff == 0 && (
            diff < 60 && "just now" ||
            diff < 120 && "1 minute ago" ||
            diff < 3600 && Math.floor( diff / 60 ) + "
minutes ago" ||
            diff < 7200 && "1 hour ago" ||
            diff < 86400 && Math.floor( diff / 3600 ) +
" hours ago") ||
        day_diff == 1 && "Yesterday" ||
        day_diff < 7 && day_diff + " days ago" ||
        day_diff < 31 && Math.ceil( day_diff / 7 ) + "
weeks ago";
}
```

- Run this example.[21]

Now that we have something to test, let's write some actual unit tests:

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
<title>Refactored date examples</title>
<script src="prettydate.js"></script>
<script>
function test(then, expected) {
```

21. http://provide.smashingmagazine.com/introduction-to-js-unit-testing-code/2-getting-somewhere.html

```
        results.total++;
        var result = prettyDate("2008-01-28T22:25:00Z",
then);
        if (result !== expected) {
                results.bad++;
                console.log("Expected " + expected + ",
but was " + result);
        }
}
var results = {
        total: 0,
        bad: 0
};
test("2008/01/28 22:24:30", "just now");
test("2008/01/28 22:23:30", "1 minute ago");
test("2008/01/28 21:23:30", "1 hour ago");
test("2008/01/27 22:23:30", "Yesterday");
test("2008/01/26 22:23:30", "2 days ago");
test("2007/01/26 22:23:30", undefined);
console.log("Of " + results.total + " tests, " +
results.bad + " failed, "
        + (results.total - results.bad) + " passed.");
        </script>
</head>
<body>

</body>
</html>
```

- Run this example.[22] (Make sure to enable a console such as Firebug or Chrome's Web Inspector.)

  This will create an ad-hoc testing framework, using only the console for output. It has no dependencies to the DOM at all, so you could just as well run it in a non-browser JavaScript environment, such as Node.js or Rhino, by extracting the code in the `script` tag to its own file.

22. http://provide.smashingmagazine.com/introduction-to-js-unit-testing-code/3-first-test.html

If a test fails, it will output the expected and actual result for that test. In the end, it will output a test summary with the total, failed and passed number of tests.

If all tests have passed, like they should here, you would see the following in the console:

Of 6 tests, 0 failed, 6 passed.

To see what a failed assertion looks like, we can change something to break it:

Expected 2 day ago, but was 2 days ago.
Of 6 tests, 1 failed, 5 passed.

While this ad-hoc approach is interesting as a proof of concept (you really can write a test runner in just a few lines of code), it's much more practical to use an existing unit testing framework that provides better output and more infrastructure for writing and organizing tests.

The QUnit JavaScript Test Suite

The choice of framework is mostly a matter of taste. For the rest of this chapter, we'll use QUnit[23] (pronounced "q-unit"), because its style of describing tests is close to that of our ad-hoc test framework.

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
<title>Refactored date examples</title>

<link rel="stylesheet" href="qunit.css" />
<script src="qunit.js"></script>
<script src="prettydate.js"></script>

<script>
test("prettydate basics", function() {
        var now = "2008/01/28 22:25:00";
        equal(prettyDate(now, "2008/01/28 22:24:30"),
"just now");
```

23. http://docs.jquery.com/Qunit

```
        equal(prettyDate(now, "2008/01/28 22:23:30"), "1
minute ago");
        equal(prettyDate(now, "2008/01/28 21:23:30"), "1
hour ago");
        equal(prettyDate(now, "2008/01/27 22:23:30"),
"Yesterday");
        equal(prettyDate(now, "2008/01/26 22:23:30"), "2
days ago");
        equal(prettyDate(now, "2007/01/26 22:23:30"),
undefined);
});
</script>
</head>
<body>
        <div id="qunit"></div>
</body>
</html>
```

- Run this example.[24]

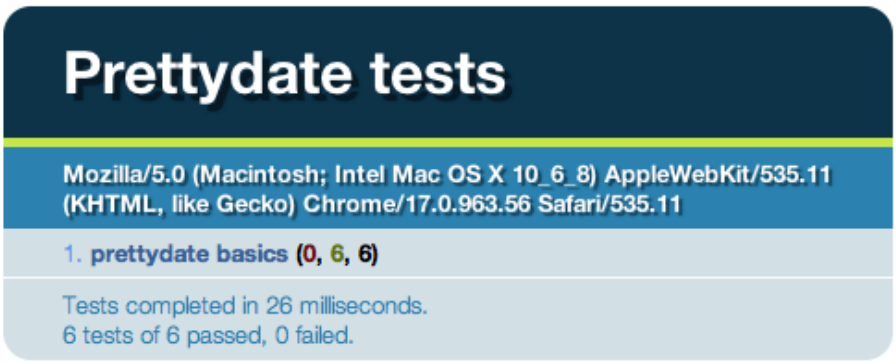
Three sections are worth a closer look here. Along with the usual HTML boilerplate, we have three included files: two files for QUnit (`qunit.css` and `qunit.js`) and the previous `prettydate.js`.

Then, there's another script block with the actual tests. The `test` method is called once, passing a string as the first argument (naming the test) and passing a function as the second argument (which will run the actual code for this test). This code then defines the `now` variable, which gets reused below, then calls the `equal` method a few times with varying arguments. The `equal` method is one of several assertions that QUnit provides. The first argument is the result of a call to `prettyDate`, with the `now` variable as the first argument and a `date` string as the second. The second argument to `equal` is the expected result. If the two arguments to `equal` are the same value, then the assertion will pass; otherwise, it will fail.

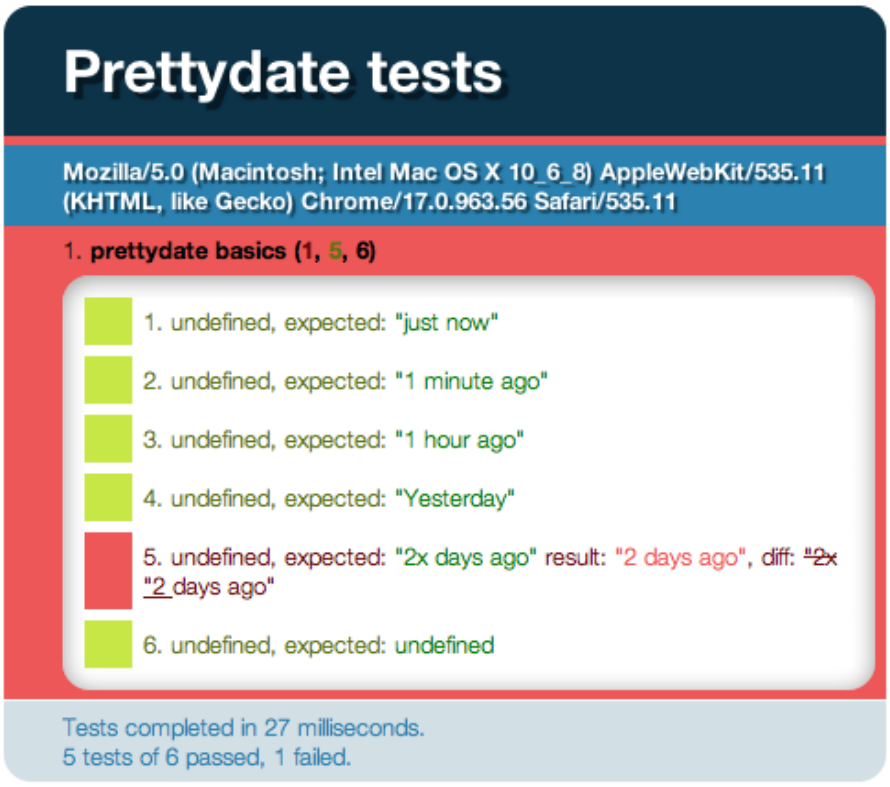
Finally, in the body element is some QUnit-specific markup. These elements are optional. If present, QUnit will use them to output the test results.

The result is this:

24. http://provide.smashingmagazine.com/introduction-to-js-unit-testing-code/4-qunit-test.html



With a failed test, the result would look something like this:



Because the test contains a failing assertion, QUnit doesn't collapse the results for that test, and we can see immediately what went wrong. Along with the output of the expected and actual values, we get a `diff` between the two, which can be useful for comparing larger strings. Here, it's pretty obvious what went wrong.

Refactoring, Stage 1

The assertions are currently somewhat incomplete because we aren't yet testing the `n weeks ago` variant. Before adding it, we should consider refactoring the test code. Currently, we are calling `prettyDate` for

each assertion and passing the `now` argument. We could easily refactor this into a custom assertion method:

```
test("prettydate basics", function() {
        function date(then, expected) {
                equal(prettyDate("2008/01/28 22:25:00",
then), expected);
        }
        date("2008/01/28 22:24:30", "just now");
        date("2008/01/28 22:23:30", "1 minute ago");
        date("2008/01/28 21:23:30", "1 hour ago");
        date("2008/01/27 22:23:30", "Yesterday");
        date("2008/01/26 22:23:30", "2 days ago");
        date("2007/01/26 22:23:30", undefined);
});
```

- Run this example.[25]

Here we've extracted the call to `prettyDate` into the `date` function, inlining the `now` variable into the function. We end up with just the relevant data for each assertion, making it easier to read, while the underlying abstraction remains pretty obvious.

Testing The DOM manipulation

Now that the `prettyDate` function is tested well enough, let's shift our focus back to the initial example. Along with the `prettyDate` function, it also selected some DOM elements and updated them, within the `window` load event handler. Applying the same principles as before, we should be able to refactor that code and test it. In addition, we'll introduce a module for these two functions, to avoid cluttering the global namespace and to be able to give these individual functions more meaningful names.

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
```

25. http://provide.smashingmagazine.com/introduction-to-js-unit-testing-code/5-qunit-test-refactored.html

```
charset=UTF-8" />
<title>Refactored date examples</title>
<link rel="stylesheet" href="qunit.css" />
<script src="qunit.js"></script>
<script src="prettydate2.js"></script>
<script>
test("prettydate.format", function() {
        function date(then, expected) {
                equal(prettyDate.format("2008/01/28
22:25:00", then), expected);
        }
        date("2008/01/28 22:24:30", "just now");
        date("2008/01/28 22:23:30", "1 minute ago");
        date("2008/01/28 21:23:30", "1 hour ago");
        date("2008/01/27 22:23:30", "Yesterday");
        date("2008/01/26 22:23:30", "2 days ago");
        date("2007/01/26 22:23:30", undefined);
});

test("prettyDate.update", function() {
        var links =
document.getElementById("qunit-fixture").getElementsByTagName("a");
        equal(links[0].innerHTML, "January 28th, 2008");
        equal(links[2].innerHTML, "January 27th, 2008");
        prettyDate.update("2008-01-28T22:25:00Z");
        equal(links[0].innerHTML, "2 hours ago");
        equal(links[2].innerHTML, "Yesterday");
});

test("prettyDate.update, one day later", function() {
        var links =
document.getElementById("qunit-fixture").getElementsByTagName("a");
        equal(links[0].innerHTML, "January 28th, 2008");
        equal(links[2].innerHTML, "January 27th, 2008");
        prettyDate.update("2008-01-28T22:25:00Z");
        equal(links[0].innerHTML, "Yesterday");
        equal(links[2].innerHTML, "2 days ago");
});
</script>
</head>
<body>
```

```
    <div id="qunit"></div>
    <div id="qunit-fixture">
        <ul>
            <li class="entry" id="post57">
                <p>blah blah blah…</p>
                <small class="extra">
                    Posted <span class="time"><a
href="/2008/01/blah/57/"
title="2008-01-28T20:24:17Z">January 28th,
2008</a></span>
                    by <span class="author"><a
href="/john/">John Resig</a></span>
                </small>
            </li>
            <li class="entry" id="post57">
                <p>blah blah blah…</p>
                <small class="extra">
                    Posted <span class="time"><a
href="/2008/01/blah/57/"
title="2008-01-27T22:24:17Z">January 27th,
2008</a></span>
                    by <span class="author"><a
href="/john/">John Resig</a></span>
                </small>
            </li>
        </ul>
    </div>
</body>
</html>
```

Here's the contents of `prettydate2.js`:

```
var prettyDate = {
        format: function(now, time){
                var date = new Date(time || ""),
                        diff = (((new
Date(now)).getTime() - date.getTime()) / 1000),
                        day_diff = Math.floor(diff /
86400);

                if (isNaN(day_diff) || day_diff < 0 ||
day_diff >= 31) {
```

```
                return;
            }

            return day_diff === 0 && (
                    diff < 60 && "just now"
||
                    diff < 120 && "1 minute
ago" ||
                    diff < 3600 &&
Math.floor( diff / 60 ) + " minutes ago" ||
                    diff < 7200 && "1 hour
ago" ||
                    diff < 86400 &&
Math.floor( diff / 3600 ) + " hours ago") ||
                day_diff === 1 && "Yesterday" ||
                day_diff < 7 && day_diff + "
days ago" ||
                day_diff < 31 && Math.ceil(
day_diff / 7 ) + " weeks ago";
        },

        update: function(now) {
            var links =
document.getElementsByTagName("a");
            for ( var i = 0; i < links.length; i++ )
{
                if (links[i].title) {
                    var date =
prettyDate.format(now, links[i].title);
                    if (date) {

links[i].innerHTML = date;
                    }
                }
            }
        }
};
```

- Run this example.[26]

The new `prettyDate.update` function is an extract of the initial example, but with the `now` argument to pass through to `prettyDate.format`.

The QUnit-based test for that function starts by selecting all `a` elements within the `#qunit-fixture` element. In the updated markup in the body element, the `<div id="qunit-fixture">…</div>` is new. It contains an extract of the markup from our initial example, enough to write useful tests against. By putting it in the `#qunit-fixture` element, we don't have to worry about DOM changes from one test affecting other tests, because QUnit will automatically reset the markup after each test.

Let's look at the first test for `prettyDate.update`. After selecting those anchors, two assertions verify that these have their initial text values. Afterwards, `prettyDate.update` is called, passing along a fixed date (the same as in previous tests). Afterwards, two more assertions are run, now verifying that the `innerHTML` property of these elements have the correctly formatted date, "2 hours ago" and "Yesterday."

Refactoring, Stage 2

The next test, `prettyDate.update, one day later`, does nearly the same thing, except that it passes a different date to `prettyDate.update` and, therefore, expects different results for the two links. Let's see if we can refactor these tests to remove the duplication.

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
<title>Refactored date examples</title>
<link rel="stylesheet" href="qunit.css" />
<script src="qunit.js"></script>
<script src="prettydate2.js"></script>
<script>
test("prettydate.format", function() {
        function date(then, expected) {
                equal(prettyDate.format("2008/01/28
22:25:00", then), expected);
        }
        date("2008/01/28 22:24:30", "just now");
        date("2008/01/28 22:23:30", "1 minute ago");
```

26. http://provide.smashingmagazine.com/introduction-to-js-unit-testing-code/6-qunit-dom.html

```
        date("2008/01/28 21:23:30", "1 hour ago");
        date("2008/01/27 22:23:30", "Yesterday");
        date("2008/01/26 22:23:30", "2 days ago");
        date("2007/01/26 22:23:30", undefined);
});

function domtest(name, now, first, second) {
        test(name, function() {
                var links =
document.getElementById("qunit-fixture").getElementsByTagName("a");
                equal(links[0].innerHTML, "January 28th,
2008");
                equal(links[2].innerHTML, "January 27th,
2008");
                prettyDate.update(now);
                equal(links[0].innerHTML, first);
                equal(links[2].innerHTML, second);
        });
}
domtest("prettyDate.update", "2008-01-28T22:25:00Z:00",
"2 hours ago", "Yesterday");
domtest("prettyDate.update, one day later",
"2008-01-29T22:25:00Z:00", "Yesterday", "2 days ago");
</script>
</head>
<body>
        <div id="qunit"></div>
        <div id="qunit-fixture">
                <ul>
                        <li class="entry" id="post57">
                                <p>blah blah blah…</p>
                                <small class="extra">
                                        Posted <span
class="time"><a href="/2008/01/blah/57/"
title="2008-01-28T20:24:17Z">January 28th,
2008</a></span>
                                        by <span
class="author"><a href="/john/">John Resig</a></span>
                                </small>
                        </li>
                        <li class="entry" id="post57">
```

```
                                    <p>blah blah blah…</p>
                                    <small class="extra">
                                            Posted <span
class="time"><a href="/2008/01/blah/57/"
title="2008-01-27T22:24:17Z">January 27th,
2008</a></span>
                                            by <span
class="author"><a href="/john/">John Resig</a></span>
                                    </small>
                            </li>
                    </ul>
            </div>
</body>
</html>
```

- Run this example.[27]

Here we have a new function called `domtest`, which encapsulates the logic of the two previous calls to test, introducing arguments for the test name, the date string and the two expected strings. It then gets called twice.

Back To The Start

With that in place, let's go back to our initial example and see what that looks like now, after the refactoring.

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html;
charset=UTF-8" />
<title>Final date examples</title>
<script src="prettydate2.js"></script>
<script>
window.onload = function() {
        prettyDate.update("2008-01-28T22:25:00Z");
};
```

27. http://provide.smashingmagazine.com/introduction-to-js-unit-testing-code/7-qunit-dom-refactored.html

```
</script>
</head>
<body>

<ul>
<li class="entry" id="post57">
        <p>blah blah blah…</p>
        <small class="extra">
                Posted <span class="time"><a href="/2008/
01/blah/57/" title="2008-01-28T20:24:17Z"><span>January
28th, 2008</span></a></span>
                by <span class="author"><a
href="/john/">John Resig</a></span>
        </small>
</li>
<li class="entry" id="post57">
        <p>blah blah blah…</p>
        <small class="extra">
                Posted <span class="time"><a href="/2008/
01/blah/57/" title="2008-01-27T22:24:17Z"><span>January
27th, 2008</span></a></span>
                by <span class="author"><a
href="/john/">John Resig</a></span>
        </small>
</li>
<li class="entry" id="post57">
        <p>blah blah blah…</p>
        <small class="extra">
                Posted <span class="time"><a href="/2008/
01/blah/57/" title="2008-01-26T22:24:17Z"><span>January
26th, 2008</span></a></span>
                by <span class="author"><a
href="/john/">John Resig</a></span>
        </small>
</li>
<li class="entry" id="post57">
        <p>blah blah blah…</p>
        <small class="extra">
                Posted <span class="time"><a href="/2008/
01/blah/57/" title="2008-01-25T22:24:17Z"><span>January
25th, 2008</span></a></span>
```

```
                by <span class="author"><a
href="/john/">John Resig</a></span>
        </small>
</li>
<li class="entry" id="post57">
        <p>blah blah blah…</p>
        <small class="extra">
                Posted <span class="time"><a href="/2008/
01/blah/57/" title="2008-01-24T22:24:17Z"><span>January
24th, 2008</span></a></span>
                by <span class="author"><a
href="/john/">John Resig</a></span>

        </small>
</li>
<li class="entry" id="post57">
        <p>blah blah blah…</p>
        <small class="extra">
                Posted <span class="time"><a href="/2008/
01/blah/57/" title="2008-01-14T22:24:17Z"><span>January
14th, 2008</span></a></span>
                by <span class="author"><a
href="/john/">John Resig</a></span>
        </small>
</li>
<li class="entry" id="post57">
        <p>blah blah blah…</p>
        <small class="extra">
                Posted <span class="time"><a href="/2008/
01/blah/57/" title="2008-01-04T22:24:17Z"><span>January
4th, 2008</span></a></span>
                by <span class="author"><a
href="/john/">John Resig</a></span>
        </small>
</li>
<li class="entry" id="post57">
        <p>blah blah blah…</p>
        <small class="extra">
                Posted <span class="time"><a href="/2008/
01/blah/57/" title="2007-12-15T22:24:17Z"><span>December
15th, 2008</span></a></span>
```

```
                by <span class="author"><a
href="/john/">John Resig</a></span>
        </small>
</li>
</ul>

</body>
</html>
```

- Run this example.[28]

For a non-static example, we'd remove the argument to `prettyDate.update`. All in all, the refactoring is a huge improvement over the first example. And thanks to the `prettyDate` module that we introduced, we can add even more functionality without clobbering the global namespace.

Conclusion

Testing JavaScript code is not just a matter of using some test runner and writing a few tests; it usually requires some heavy structural changes when applied to code that has been tested only manually before. We've walked through an example of how to change the code structure of an existing module to run some tests using an ad-hoc testing framework, then replacing that with a more full-featured framework to get useful visual results.

QUnit itself has a lot more to offer, with specific support for testing asynchronous code such as timeouts, AJAX and events. Its visual test runner helps to debug code by making it easy to rerun specific tests and by providing stack traces for failed assertions and caught exceptions. For further reading, check out the QUnit Cookbook[29]. ❧

28. http://provide.smashingmagazine.com/introduction-to-js-unit-testing-code/8-endstate.html
29. http://qunitjs.com/cookbook

Journey Through The JavaScript MVC Jungle

BY ADDY OSMANI

When writing a Web application from scratch, it's easy to feel like we can get by simply by relying on a DOM[30] manipulation library (like jQuery[31]) and a handful of utility plugins. The problem with this is that it doesn't take long to get lost in a nested pile of jQuery callbacks and DOM elements without any real structure in place for our applications.

In short, we're stuck with spaghetti code[32]. Fortunately there are modern JavaScript frameworks that can assist with bringing structure and organization to our projects, improving how easily maintainable they are in the long-run.

What Is MVC, Or Rather MV?*

These modern frameworks provide developers an easy path to organizing their code using variations of a pattern known as MVC[33] (Model-View-Controller). MVC separates the concerns in an application down into three parts:

- **Models** represent the domain-specific knowledge and data in an application. Think of this as being a 'type' of data you can model—like a User, Photo or Note. Models should notify anyone observing them about their current state (e.g Views).
- **Views** are typically considered the User-interface in an application (e.g your markup and templates), but don't have to be. They should know about the existence of Models in order to observe them, but don't directly communicate with them.
- **Controllers** handle the input (e.g clicks, user actions) in an application and Views can be considered as handling the output. When a Controller updates the state of a model (such as editing the caption on a Photo), it doesn't directly tell the View. This is what the observing nature of the View and Model relationship is for.

30. https://developer.mozilla.org/en/DOM/About_the_Document_Object_Model
31. http://jquery.com
32. http://en.wikipedia.org/wiki/Spaghetti_code
33. http://addyosmani.com/resources/essentialjsdesignpatterns/book/#detailmvcmvp

JavaScript 'MVC' frameworks that can help us structure our code don't always strictly follow the above pattern. Some frameworks will include the responsibility of the Controller in the View (e.g Backbone.js[34]) whilst others add their own opinionated components into the mix as they feel this is more effective.

For this reason we refer to such frameworks as following the MV* pattern, that is, you're likely to have a View and a Model, but more likely to have something else also included.

Note: There also exist variations of MVC known as MVP (Model-View-Presenter) and MVVM (Model-View ViewModel). If you're new to this and feel it's a lot to take in, don't worry. It can take a little while to get your head around patterns, but I've written more about the above patterns in my online book Learning JavaScript Design Patterns[35] in case you need further help.

When Do You Need A JavaScript MV Framework?*

When building a single-page application using JavaScript, whether it involves a complex user interface or is simply trying to reduce the

34. http://backbonejs.org
35. http://addyosmani.com/resources/essentialjsdesignpatterns/book/#detailmvcmvp

number of HTTP requests required for new Views, you will likely find yourself inventing many of the pieces that make up an MV* framework like Backbone or Ember.

At the outset, it isn't terribly difficult to write an application framework that offers *some* opinionated way to avoid spaghetti code, however to say that it is equally as trivial to write something of the standard of Backbone would be a grossly incorrect assumption.

There's a lot more that goes into structuring an application than tying together a DOM manipulation library, templating and routing. Mature MV* frameworks typically not only include many of the pieces you would find yourself writing, but also include solutions to problems you'll find yourself running into later on down the road. This is a timesaver that you shouldn't underestimate the value of.

So, where will you likely need an MV* framework and where won't you?

If you're writing an application that will likely only be communicating with an API or back-end data service, where much of the heavy lifting for viewing or manipulating that data will be occurring in the browser, you may find a JavaScript MV* framework useful.

Good examples of applications that fall into this category are GMail[36] and Google Docs[37]. These applications typically download a single payload containing all the scripts, stylesheets and markup users need for common tasks and then perform a lot of additional behavior in the background. It's trivial to switch between reading an email or document to writing one and you don't need to ask the application to render the whole page again at all.

If, however, you're building an application that still relies on the server for most of the heavy-lifting of Views/pages and you're just using a little JavaScript or jQuery to make things a little more interactive, an MV framework may be overkill. There certainly are complex Web applications where the partial rendering of views can* be coupled with a single-page application effectively, but for everything else, you may find yourself better sticking to a simpler setup.

The Challenge Of Choice: Too Many Options?

The JavaScript community has been going through something of a renaissance over the last few years, with developers building even larger and more complex applications with it as time goes by. The language still greatly differs from those more classic Software engineers are used

36. http://gmail.com
37. http://docs.google.com

to using (C++, Java) as well as languages used by Web developers (PHP, Python, .Net etc). This means that in many cases we are borrowing concepts of how to structure applications from what we have seen done in the past in these other languages.

In my talk "Digesting JavaScript MVC: Pattern Abuse or Evolution[38]", I brought up the point that there's currently too much choice when it comes to what to use for structuring your JavaScript application. Part of this problem is fueled by how different JavaScript developers interpret how a scalable JavaScript application should be organized—MVC? MVP? MVVM? Something else? This leads to more frameworks being created with a different take on MV* each week and ultimately more noise because we're still trying to establish the "right way" to do things, if that exists at all. Many developers believe it doesn't.

We refer to the current state of new frameworks frequently popping up as 'Yet Another Framework Syndrome' (or YAFS). Whilst innovation is of course something we should welcome, YAFS can lead to a great deal of confusion and frustration when developers just want to start writing an app but don't want to manually evaluate 30 different options in order to select something maintainable. In many cases, the differences between some of these frameworks can be very subtle if not difficult to distinguish.

TodoMVC: A Common Application For Learning And Comparison

There's been a huge boom in the number of such MV* frameworks being released over the past few years.

Backbone.js[39], Ember.js[40], AngularJS[41], Spine[42], CanJS[43] ... The list of new and stable solutions continues to grow each week and developers can quickly find themselves lost in a sea of options. From minds who have had to work on complex applications that inspired these solutions (such as Yehuda Katz[44] and Jeremy Ashkenas[45]), there are many strong contenders for what developers should consider using. The question is, what to use and how do you choose?

---

38. http://addyosmani.com/blog/digesting-javascript-mvc-pattern-abuse-or-evolution/
39. http://backbonejs.org
40. http://emberjs.com
41. http://angularjs.com
42. http://spinejs.com
43. http://canjs.us
44. http://github.com/wycats
45. http://github.com/jashkenas

We understood this frustration and wanted to help developers simplify their selection process as much as possible. To help solve this problem, we created TodoMVC[46]—a project which offers the same Todo application implemented in most of the popular JavaScript MV* frameworks of today—think of it as speed dating for frameworks. Solutions look and feel the same, have a common feature set, and make it easy for us to compare the syntax and structure of different frameworks, so we can select the one we feel the most comfortable with or at least, narrow down our choices.

This week we're releasing a **brand new** version of TodoMVC[47], which you can find more details about lower down in the apps section.

In the near future we want to take this work even further, providing guides on how frameworks differ and recommendations for which options to consider for particular types of applications you may wish to build.

46. http://todomvc.com
47. http://www.todomvc.com

Our Suggested Criteria For Selecting A Framework

Selecting a framework is of course about more than simply comparing the Todo app implementations. This is why, once we've filtered down our selection of potential frameworks to just a few, it's recommend to spend some time doing a little due diligence. The framework we opt for may need to support building non-trivial features and could end up being used to maintain the app for years to come.

- **What is the framework really capable of?**
  Spend time reviewing both the source code of the framework and official list of features to see how well they fit with your requirements. There will be projects that may require modifying or extending the underlying source and thus make sure that if this might be the case, you've performed due diligence on the code.

- **Has the framework been proved in production?**
  i.e Have developers actually built and deployed large applications with it that are publicly accessible? Backbone has a strong portfolio of these (SoundCloud, LinkedIn) but not all frameworks do. Ember is used in number of large apps, including the user tools in Square. JavaScript-MVC has been used to power applications at IBM amongst other places. It's not only important to know that a framework works in production, but also being able to look at real world code and be inspired by what can be built with it.

- **Is the framework mature?**
  We generally recommend developers don't simply "pick one and go with it". New projects often come with a lot of buzz surrounding their releases but remember to take care when selecting them for use on a production-level app. You don't want to risk the project being canned, going through major periods of refactoring or other breaking changes that tend to be more carefully planned out when a framework is mature. Mature projects also tend to have more detailed documentation available, either as a part of their official or community-driven docs.

- **Is the framework flexible or opinionated?**
  Know what flavor you're after as there are plenty of frameworks available which provide one or the other. Opinionated frameworks lock (or suggest) you to do things in a specific way (theirs). By design they are limiting, but place less emphasis on the developer having to figure out how things should work on their own.

- **Have you really played with the framework?**
  Write a small application without using frameworks and then attempt to refactor your code with a framework to confirm whether it's easy to

work with or not. As much as researching and reading up on code will influence your decision, it's equally as important to write actual code using the framework to make sure you're comfortable with the concepts it enforces.

- **Does the framework have a comprehensive set of documentation?**
  Although demo applications can be useful for reference, you'll almost always find yourself consulting the official framework docs to find out what its API supports, how common tasks or components can be created with it and what the gotchas worth noting are. Any framework worth it's salt should have a detailed set of documentation which will help guide developers using it. Without this, you can find yourself heavily relying on IRC channels, groups and self-discovery, which can be fine, but are often overly time-consuming when compared to a great set of docs provided upfront.

- **What is the total size of the framework, factoring in minification, gzipping and any modular building that it supports?**
  What dependencies does the framework have? Frameworks tend to only list the total filesize of the base library itself, but don't list the sizes of the librarys dependencies. This can mean the difference between opting for a library that initially looks quite small, but could be relatively large if it say, depends on jQuery and other libraries.

- **Have you reviewed the community around the framework?**
  Is there an active community of project contributors and users who would be able to assist if you run into issues? Have enough developers been using the framework that there are existing reference applications, tutorials and maybe even screencasts that you can use to learn more about it?

Dojo And Rise Of The JavaScript Frameworks

As many of us know, the Dojo toolkit[48] was one of the first efforts to provide developers a means to developing more complex applications and some might say it in-part inspired us to think more about the needs of non-trivial applications. I sat down to ask Dojos Dylan Schiemann[49], Kitson Kelly, and James Thomas what their thoughts were on the rise of JavaScript MV* frameworks.

48. http://dojotoolkit.org
49. http://dylanschiemann.com

Q: Didn't Dojo already solve all of this? Why hasn't it been the dominent solution for developers wishing to build more structured (and more non-trivial) applications?

Years ago, while the JavaScript landscape evolved from adding simple Ajax and chrome to a page, Dojo was evangelizing a "toolkit" approach to building complex Web applications.

Many of those features were way ahead of most developers needs. With the emergence of the browser as the dominant application platform, many of the innovations pioneered in The Dojo Toolkit now appear in newer toolkits. MVC was just another package that Dojo has provided for quite some time, along with modular code packages, OO in JS, UI widgets, cross-browser graphics, templating, internationalization, accessibility, data stores, testing frameworks, a build system and much, much more.

JavaScript libraries shouldn't end at "query", which is why Dojo, early on, focussed on completing the picture for enterprise grade application development. This is the same focus that is has today with MVC, it's just another "tool in the arsenal".

Why is Dojo not the dominant toolkit? Its goal was never to be the only choice. The goal was to provide an open collection of tools that could be used with anything else, within projects, and liberally copied into other work as well. Dojo was criticized for being slow and even after that was addressed, it was criticized for being slow. Trying to shake that perception is challenging. It is very hard to document a feature-rich toolkit. There are 175 sub-packages in Dojo 1.8 and over 1,400 modules.

That is not only a challenge from a documentation purpose, it also means that there isn't one thing that Dojo does. Which is good if you are building software, but very difficult when you are starting out trying to figure out where to start. These are all things we have been trying to work on for Dojo 1.8, in the form of tutorials and significantly improved documentation.

Q: Why should developers still consider Dojo and what ideas do you have lined up for the future of the project? I hear 1.8 will be another major milestone.

In Dojo 1.8, dojox/mvc takes another step towards full maturity. There has been a lot of investment in time, effort, testing and community awareness into the package. It focuses on providing an MVC model that leverages the rest of Dojo. Coupled with dojox/app, an application framework that is designed to make it easier to build rich applications

across desktop and mobile, it makes a holistic framework for creating a client-side application.

In the typical Dojo way, this is just one of many viable ways in which to build applications with Dojo.

In 1.8, not only does the MVC sub-module become more mature, it is built upon a robust framework. It doesn't just give you markup language to create your views, express your models or develop a controller. It is far more then just wiring up some controls to a data source. Because it is leveraging the rest of Dojo, you can draw in anything else you might need.

In Dojo 2.0 we will be looking to take modularity to a new level, so that it becomes even easier to take a bit of this and a bit of that and string it all together. We are also exploring the concepts of isomorphism, where it should be transparent to the end-user where your code is being executed, be it client side or server side and that ultimately it should be transparent to the developer.

The TodoMVC Collection

In our brand new release, Todo implementations now exist for the most popular frameworks with a large number of other commonly used frameworks being worked on in Labs. These implementations have gone through a lot of revision, often taking on board best practice tips and suggestions from framework authors, contributors and users from within the community.

Following on from comments previously made by Backbone.js author Jeremey Ashkenas and Yehuda Katz, TodoMVC now also offers consistent implementations based on an official application specification as well as routing (or state management).

We don't pretend that more complex learning applications aren't possible (they certainly are), but the simplicity of a Todo app allows developers to review areas such as code structure, component syntax and flow, which we feel are enough to enable a comparison between frameworks and prompt further exploration with a particular solution or set of solutions.

Our applications include:

- Backbone.js[50]
- Ember.js[51]
- AngularJS[52]
- Spine.js[53]

50. http://documentcloud.github.com/backbone
51. http://emberjs.com
52. http://angularjs.org
53. http://spinejs.com

- KnockoutJS[54] (MVVM)
- Dojo[55]
- YUI[56]
- Batman.js[57]
- Closure[58]
- Agility.js[59]
- Knockback.js[60]

For those interested in AMD versions:

- Backbone.js[61] + RequireJS[62] (using AMD)
- Ember.js[63] + RequireJS[64] (using AMD)

And our Labs include:

- CanJS[65]
- Maria.js[66]
- cujo.js[67]
- Meteor[68]
- SocketStream[69] + jQuery[70]
- Ext.js[71]

---

54. http://knockoutjs.com
55. http://dojotoolkit.org
56. http://yuilibrary.com
57. http://batmanjs.org
58. http://code.google.com/closure/library/
59. http://agilityjs.com
60. http://kmalakoff.github.com/knockback
61. http://documentcloud.github.com/backbone
62. http://requirejs.org
63. http://emberjs.com
64. http://requirejs.org
65. http://canjs.us
66. https://github.com/petermichaux/maria
67. http://cujojs.github.com
68. http://meteor.com
69. http://www.socketstream.org
70. http://jquery.com

- Sammy.js[72]
- JavaScriptMVC[73]
- Google Web Toolkit[74]
- TroopJS[75]
- Stapes.js[76]
- soma.js[77]
- DUEL[78]
- Fidel[79]
- Olives[80]
- PlastronJS[81]
- Dijon[82]
- rAppid.js[83]
- Broke[84]
- o_O[85]
- Fun[86]
- AngularJS[87] + RequireJS[88] (using AMD)

Note: We've implemented a version of our Todo application using just JavaScript[89] and another using primarily jQuery conventions[90]. As you

---

71. http://www.sencha.com/products/extjs
72. http://sammyjs.org
73. http://javascriptmvc.com
74. https://developers.google.com/web-toolkit/
75. https://github.com/troopjs
76. http://hay.github.com/stapes
77. http://somajs.github.com/somajs
78. https://bitbucket.org/mckamey/duel/wiki/Home
79. https://github.com/jgallen23/fidel
80. https://github.com/flams/olives
81. https://github.com/rhysbrettbowen/PlastronJS
82. https://github.com/creynders/dijon-framework
83. http://www.rappidjs.com
84. https://github.com/brokenseal/broke
85. http://weepy.github.com/o_O
86. https://github.com/marcuswestin/fun
87. http://angularjs.org
88. http://requirejs.org
89. https://github.com/addyosmani/todomvc/blob/master/vanilla-examples/vanillajs/js/app.js

can see, whilst these applications are functionally equivalent to something you might write with an MVC framework, there's no separation of concerns and the code becomes harder to read and maintain as the codebase grows.

We feel honored that over the past year, some framework authors have involved us in discussions about how to improve their solutions, helping bring our experience with a multitude of solutions to the table. We've also slowly moved towards TodoMVC being almost a defacto app that new frameworks implement and this means it's become easier to make initial comparisons when you're reviewing choices.

Frameworks: When To Use What?

To help you get started with narrowing down frameworks to explore, we would like to offer the below high-level framework summaries which we hope will help steer you towards a few specific options to try out.

I want something flexible which offers a minimalist solution to separating concerns in my application. It should support a persistence layer and RESTful sync, models, views (with controllers), event-driven communication, templating and routing. It should be imperative, allowing one to update the View when a model changes. I'd like some decisions about the architecture left up to me. Ideally, many large companies have used the solution to build non-trivial applications. As I may be building something complex, I'd like there to be an active extension community around the framework that have already tried addressing larger problems (Marionette[91], Chaplin[92], Aura[93], Thorax[94]). Ideally, there are also scaffolding tools (grunt-bbb[95], brunch[96]) available for the solution. **Use Backbone.js.**

I want something that tries to tackle desktop-level application development for the web. It should be opinionated, modular, support a variation of MVC, avoid the need to wire everything in my application together manually, support persistence, computed properties and have auto-updating (live) templates. It should support proper state management rather than the manual routing solution many other frameworks advocate being used. It should also come with extensive docs and of

90. https://github.com/addyosmani/todomvc/blob/master/architecture-examples/jquery/js/app.js
91. https://github.com/derickbailey/backbone.marionette
92. https://github.com/chaplinjs/chaplin
93. https://github.com/addyosmani/backbone-aura/
94. https://github.com/walmartlabs/thorax
95. https://github.com/backbone-boilerplate/grunt-bbb
96. http://brunch.io/

course, templating. It should also have scaffolding tools available (ember.gem, ember for brunch). **Use Ember.js.**

I want something more lightweight which supports live-binding templates, routing, integration with major libraries (like jQuery and Dojo) and is optimized for performance. It should also support a way to implement models, views and controllers. It may not be used on as many large public applications just yet, but has potential. Ideally, the solution should be built by people who have previous experience creating many complex applications. **Use CanJS.**

I want something declarative that uses the View to derive behavior. It focuses on achieving this through custom HTML tags and components that specify your application intentions. It should support being easily testable, URL management (routing) and a separation of concerns through a variation of MVC. It takes a different approach to most frameworks, providing a HTML compiler for creating your own DSL in HTML. It may be inspired by upcoming Web platform features such as Web Components and also has its own scaffolding tools available (angular-seed). **Use AngularJS.**

I want something that offers me an excellent base for building large scale applications. It should support a mature widget infrastructure, modules which support lazy-loading and can be asynchronous, simple integration with CDNs, a wide array of widget modules (graphics, charting, grids, etc) and strong support for internationalization (i18n, l10n). It should have support for OOP, MVC and the building blocks to create more complex architectures. **Use Dojo.**

I want something which benefits from the YUI extension infrastructure. It should support models, views and routers and make it simple to write multi-view applications supporting routing, View transitions and more. Whilst larger, it is a complete solution that includes widgets/components as well as the tools needed to create an organized application architecture. It may have scaffolding tools (yuiproject), but these need to be updated. **Use YUI.**

I want something simple that values asynchronous interfaces and lack any dependencies. It should be opinionated but flexible on how to build applications. The framework should provide bare-bones essentials like model, view, controller, events, and routing, while still being tiny. It should be optimized for use with CoffeeScript and come with comprehensive documentation. **Use Spine.**

I want something that will make it easy to build complex dynamic UIs with a clean underlying data model and declarative bindings. It should automatically update my UI on model changes using two-way bindings and support dependency tracking of model data. I should be able to use it with whatever framework I prefer, or even an existing

app. It should also come with templating built-in and be easily extensible. **Use KnockoutJS.**

I want something that will help me build simple Web applications and websites. I don't expect there to be a great deal of code involved and so code organisation won't be much of a concern. The solution should abstract away browser differences so I can focus on the fun stuff. It should let me easily bind events, interact with remote services, be extensible and have a huge plugin community. **Use jQuery.**

What Do Developers Think About The Most Popular Frameworks?

As part of our research into MV* frameworks for TodoMVC and this chapter, we decided to conduct a survey to bring together the experiences of those using these solutions. We asked developers what framework they find themselves using the most often and more importantly, why they would recommend them to others. We also asked what they felt was still missing in their project of choice.

We've grouped some of the most interesting responses below, by framework.

EMBER.JS

Pros: The combination of live templates and observable objects has changed the way I write JavaScript. It can be a bit much to wrap your head around at first, but you end up with a nice separation of responsibility. I found that once I have everything set up, adding fairly complex features only takes a couple lines of code. Without Ember, these same features would've been hellish to implement.
Cons: Ember has yet to reach 1.0. Many things are still in flux, such as the router and Ember data. The new website is very helpful, but there's still not as much documentation for Ember as there is for other frameworks, specifically Backbone. Also, with so much magic in the framework, it can be a little scary. There's the fear that if something breaks you won't be able to figure out exactly why. Oh, and the error messages that ember gives you often suck.

Pros:
The key factors: a) Features that let me avoid a lot of boilerplate (bindings, computer properties, view layer with the cool handlebars). b) the core team: I'm a Rails developer and know the work of Yehuda Katz. I trust the guy.
Cons: Documentation. It's really sad that Ember doesn't have good doc-

umentation, tutorials, screencast like Backbone, Angular or other frameworks. Right now, we browse the code looking for docs which isn't ideal.

Pros: Convention over configuration. Ember makes so many small decisions for you it's by far the easiest way to build a client-side application these days.
Cons: The learning curve. It is missing the mass of getting started guides that exist for other frameworks like Backbone, this is partly because of the small community, but I think more because of the state of flux the codebase is in pre-1.0.

Pros: Simplicity, bindings, tight integration with Handlebars, ease of enabling modularity in my own code.
Cons: I'd like to have a stable integration with ember-data, and integrated localStorage support synced with a REST API, but hey that's fantasy that one day will surely come true ;-)

BACKBONE.JS

Pros: Simplicity—only 4 core components (Collection, Model, View, Router). Huge community (ecosystem) and lots of solutions on StackOverflow. Higher order frameworks like Marionette or Vertebrae with lots of clever code inside. Somebody might like "low-levelness"—need to write lots of boilerplate code, but get customized application architecture.
Cons: I don't like how extend method works—it copies content of parent objects into new one. Prototypal inheritance FTW. Sometime I miss real world scenarios in docs examples. Also there is a lot of research needed to figure out how to build a bigger app after reading the TODO tutorial.

I'm missing official AMD support in projects from DocumentCloud (BB, _). [Note: this shouldn't be an issue with the new RequireJS `shim()` method in RequireJS 2.0].

Pros: After the initial brain-warp of understanding how Backbone rolls, it is incredibly useful. Useful as in, well supported, lightweight, and constantly updated in a valid scope. Ties in with natural friends Underscore, jQuery/Zepto, tools that most of my studio's projects would work with.
Cons: The amount of tutorials on how to do things with Backbone is inconsistent and at different periods of Backbones lifespan. I've asked other devs to have a look at Backbone, and they would be writing code

for v0.3. Un-aware. Whilst not a problem Backbone can fix itself, it is certainly a major dislike associated with the framework.

I suppose in theory, you could apply this to anything else, but, Backbone is a recurrent one in my eyes. Hell, I've even seen month old articles using ancient Backbone methods and patterns.

Whatever dislikes I would have on the framework strictly itself, has been rectified by the community through sensible hacks and approaches. For me, that is why Backbone is great, the community backing it up.

Pros: Provides just enough abstraction without unreasonable opinions—enabling you to tailor it to the needs of the project.
Cons: I would re-write (or possibly remove) Backbone.sync. It has baked in assumptions of typical client-initiated HTTP communications, and doesn't adapt well to the push nature of WebSockets.

Pros: It's extremely easy to get into, offering a nice gateway to MV* based frameworks. It's relatively customizable and there are also tons of other people using it, making finding help or support easy.
Cons: The fact that there's no view bindings by default (although you can fix this). Re-rendering the whole view when a single property changes is wasteful.

The RESTful API has a lot of positives, but the lack of bulk-saving (admittedly a problem with REST itself, but still) and the difficulty in getting different URI schemes to work on different types of operations sucks.

ANGULARJS

Pros:

a) 2-way data binding is incredibly powerful. You tend to think more about your model and the state that it is in instead of a series of events that need to happen. The model is the single source of truth.

b) Performance. AngularJS is a small download. It's templating uses DOM nodes instead of converting strings into DOM nodes and should perform better.

c) If you are targeting modern browsers and/or are a little careful, you can drop jQuery from your dependencies too.
Cons: I'd like to be able to specify transitions for UI state changes that propgate from a model change. Specifically for elements that use ng-show or ng-hide I'd like to use a fade or slide in in an easy declarative way.

Pros: It's very intuitive, has excellent documentation. I love their data binding approach, HTML based views, nested scopes. I switched from

Backbone/Thorax to Angular and never looked back. A new Chrome extension Batarang integrates with Chrome Developer's Tools and provides live access the Angular data structures.
Cons: I'd like to have a built-in support to such functions as drag'n'drop, however this can be added using external components available on GitHub. I'd also like to see more 3rd party components available for reuse. I think it's just a matter of time for the ecosystem around AngularJS to get more mature and then these will be available just like they are in communities like jQuery.

Pros: It minimizes drastically the boilerplate code, allows for nice code reuse through components, extends the HTML syntax so that many complex features end up being as simple as applying a directive (attribute) in the HTML, and is super-easily testable thanks to a full commitment to dependency injection.
You can write a non-trivial app without jQuery or without directly manipulating the DOM. That's quite a feat.
Cons: Its learning curve is somewhat steeper than Backbone (which is quite easy to master), but the gain is appreciative. Documentation could be better.

KNOCKOUTJS

Pros: I don't necessarily use it *all* the time, but KnockoutJS is just fantastic for single page applications. Extremely easy subscribing to live sorting; much better API for so called "collection views" in Backbone using observable arrays. And custom event on observables for effects, etc.
Cons: Feel like the API is quite hard to scale, and would probably prefer to wrangle Backbone on the bigger applications. (But that's also partially due to community support).

Pros: I like the data binding mechanism and feel very comfortable using it. In particular I like how they have replaced templates with control flow binding.
Cons: I don't like that there is no guidance or best practice in terms of application structure. Aside from having a view model, the framework doesn't help you in defining a well structured view model. It's very easy to end up with a large unmaintainable function.

DOJO

Pros: Syntactically, Dojo is very simple. It allows for dynamic and robust builds, with the initial loader file being as low as 6k in some cases. It is AMD compatible, making it extremely portable, and comes out-of-

the-box with a ton of features ranging from basic dom interactions to complex SVG, VML, and canvas functionality. The widget system, Dijit, is unmatched in it's ease-of-use and ability to be extended. It's a very well-rounded and complete toolkit.
Cons: The dojo/_base/declare functionality is not 100% strict mode compliant and there is currently some overhead due to backwards compatibility, though this will mostly go away in the Dojo 2.0 release.

Pros: Good components : tabs, datagrid, formManager... Renders the same cross browser. AMD compliant. Easy to test with mocks.Integrates well with other frameworks thks to amd (I ll integrate with JMVC)
Cons: Default design for components out of fashion. Not fully html5. So-so documentation
Poor templating system (no auto binding).

YUI

Pros: YUI3 is a modular and use-at-will type of component library which includes all of the goodies of Backbone and more. It even (in my opinion) improves upon some of the concepts in Backbone by de-coupling some things (i.e. attribute is a separate module that can be mixed into any object – the event module can be mixed in similarly).
Cons: I'd love to see YUI3 support some of the auto-wiring (optional) of Ember. I think that is really the big win for Ember; otherwise, I see YUI3 as a superior component library where I can cherry-pick what I need. I'd also like to see a more AMD-compatible module loader. The loader today works very well; however, it would be nicer if I could start a new projects based on AMD modules and pull in certain YUI3 components and other things from other places that are also using AMD.

JAVASCRIPTMVC

Pros: Has all tools included, just need to run commands and start building. I have used for the last 6 months and it's been really good.
Cons: The only thing I would do is to speed up development of the next version. Developers are aware of problems and fixing issues but its going to be another ¾ months before some issues I want fixed are addressed, but then I could probably patch and do a pull request.

MARIA

Pros: Because Maria is a pure MVC framework that is focused on being just an MVC framework. No more and no less. Its clean and simple.
Cons: A little more usage documentation outside of the source code,

plus a few more test cases. A tutorial that drives home the real use of MVC with Maria would be good too.

CUJO.JS

Pros: Real apps almost never fit perfectly into an MV* box, and the most important stuff is often outside the box. With cujo.js, you define the box.

Yes, cujo.js has high-level MV*-like features for creating views, models, controllers, etc., but every app is different, and no framework can ever be a 100% solution. Rather than try to be all things, cujo.js also provides lower level tools, architectural plumbling, and a rich plugin system that can even be used to integrate and extend other MV* frameworks.

Create the architecture that best suits your application, rather than constraining your app to fit inside someone else's predefined architecture.

Cons: The broader JavaScript community is totally unprepared and untrained to take on large-scale applications. Most of us don't even know that design patterns and architectural patterns exist.

Since cujo.js is so different from other frameworks, it needs more than a simple API reference and code snippets. Without tutorials, educational materials, and step-by-step examples, cujo.js might look strange and overwhelming to the untrained eye but documentation is supposed to be coming soon.

EXTJS

Pros: I think ExtJS works best in combination with Ext Designer. It gives it an edge beyond the other GUI frameworks by letting non-programmers mock up the UI so programmers can fill in the blanks. I think comparing it to MVC frameworks like Backbone doesn't do it justice – its strength lies in creating rich GUIs, not lean Web apps.

For rich, commercial back-office applications I think ExtJS remains the best choice when it comes to JavaScript solutions (i.e. not GWT etc). For public-facing Web apps I'd rather have something that gives me more control over the markup (and ideally something that degrades gracefully).

Cons: It has a steeper learning curve than many of the other modern structural frameworks. One can argue that if you're investing in ExtJS for the long-term this time spent learning will pay off, however I think solutions like it should aim to better minimize the time it takes to train teams up in using it.

Pros: I think a big feature of ExtJS 4 is that it throws you into the MVC mindset and the preferred filesystem structure right from the bat. With Dojo the initial tutorials seem to be mostly about augmenting existing websites whereas ExtJS assumes you're starting from scratch.

Using ExtJS doesn't really "feel" like you're dealing with HTML at all. The component library is rich enough to let you go a long way without touching more HTML than what is needed to bootstrap your app.

It'd be interesting to see how both compare when Web components become more widely supported. This would finally allow manipulating the DOM without being afraid of breaking any widgets or causing your app's internal state to become inconsistent.

Cons: The licensing is considered restrictive and difficult to understand by some. More people would be investing in ExtJS if it was clearer what the upfront and long-term costs of using it are. This isn't a concern with some other structural solutions but probably isn't as much a worry for larger businesses.

Pros: ExtJS is a fantastic package for rapidly building out RIAs for internal use. I for one, love to build with HTML and JavaScript, and for me there's great satisfaction in mucking around at that level. Even though ExtJS makes it feel like you're not really working with HTML it still offers a great deal of power, especially if you're using it to create a complex UI.

Cons: That said...I absolutely agree that it's very heavy and I don't think I'd recommend it for an external facing Web application. My biggest beef with the package overall is actually that it's more of a PITA to test with than I'd would like. Our tester actually ended up switching to Sikuli because it was becoming too much of a battle trying to work with it in Selenium.

BATMAN

Pros: It has a great and easy to use view bindings system. Plays with Rails very nicely and is all about convention over configuration.

Cons: The documentation could be a lot better and I feel Shopify won't be adding the features that they say that they will.

Don't Be Afraid To Experiment

Whilst it's unlikely for a developer to need to learn how to use more than a handfull of these frameworks, I do encourage exploration of those you're unfamiliar with. There's more than mountain of interesting facts and techniques that can be learned in this process.

In my case: I discovered that Batman.js required the least hand-written lines of code for an implementation. I'm neither a frequent CoffeeScript nor Batman.js user but that in itself gave me some food for thought. Perhaps I could take some of what made this possible and bring it over to the frameworks I do use. Or, maybe I'd simply use Batman.js in a future project if I found the community and support around it improved over time.

Regardless of whether you end up using a different solution, at the end of the day all you have to gain from exploration is more knowledge about what's out there.

Going Beyond MV Frameworks*

Whilst the MV* family of patterns are quite popular for structuring applications, they're limited in that they don't address any kind of application layer, communication between Views, services that perform work or anything else. Developers may thus find that they sometimes need to explore beyond just MVC—there are times when you absolutely need to take what they have to offer further.

We reached out to developers that have been taking MVC further with their own patterns or extensions for existing frameworks to get some insights on where you need something more.

> *"In my case, I needed something Composite. I noticed that there were patterns in Backbone apps where developers realized there was a need for an object that coordinated various parts of an application. Most of the time, I've seen developers try to solve this using a Backbone construct (e.g a View), even when there isn't really a need for it. This is why I instead explored the need for an Application Initializer*[97]*.*
>
> *I also found that MVC didn't really describe a way to handle regions of a page or application. The gist of region management*[98] *is that you could define a visible area of the screen and build out the most basic layout for it without knowing what content was going to be displayed in it at runtime.*
>
> *I created solutions for region management, application initialization and more in my extension project Marionette. It's one of a number of solutions that extend upon a framework (or architecture pattern) that developers end up needing when they're building single-page applications that are relatively complex.*

97. http://lostechies.com/derickbailey/2011/12/16/composite-javascript-applications-with-backbone-and-backbone-marionette/
98. http://lostechies.com/derickbailey/2011/12/12/composite-js-apps-regions-and-region-managers/

There's even a TodoMVC Marionette app[99] available for anyone wishing to compare the standard Backbone application with one that goes beyond just MV."*

— Derick Bailey, author of Marionette

"While a good portion of problems can be decomposed into JavaScript MVC, there are some which simply cannot. For example, an application consumes a third party API at runtime, but is not given any information as to how the data will be structured.

I spent almost a year trying to solve that very problem, but eventually I came to the realization that shoehorning it into MV was not a viable solution. I was dealing with an "amorphous model" and that's where it all fell apart. In other words, if you don't have a well-defined model, most modern JavaScript frameworks can't help you.*

That's where Core J2EE Patterns[100] come in. I got turned on to them while reading PHP Objects, Patterns, and Practice[101] by Matt Zandstra, and I'm glad I did! The J2EE Patterns basically outline a request-driven process, where the URL drives the behavior of the application. In a nutshell, a request is created, modified, and then used to determine the view to render.

I've expanded on my experiences with request driven Javascript applications and J2EE patterns[102] for anyone who would like to learn more."

— Dustin Boston, co-author, Aura

Conclusions

While there are several choices for what to use for structuring your JavaScript Web applications these days, it's important to be **diligent** in the selection process – spend time thoroughly evaluating your options in order to make a decision which results in sustainable, **maintainable** code. Framework diversity fosters innovation, while too much similarity just creates noise.

Projects like TodoMVC can help narrow down your selections to those you feel might be the most interesting or most comfortable for a particular project. Remember to take your time choosing, don't feel too constrained by using a specific pattern and keep in mind that it's completely acceptable to build on the solution you select to best fit the needs of your application.

99. https://github.com/derickbailey/todomvc/tree/marionette
100. http://java.sun.com/blueprints/corej2eepatterns/Patterns/
101. http://www.amazon.com/Objects-Patterns-Practice-Matt-Zandstra/dp/1590599098
102. http://dblogit.com/archives/3895

Experimenting with different frameworks will also give you different views on how to solve common problems which will in turn make you a better programmer.

Thanks to my fellow TodoMVC team-member Sindre Sorhus[103] for his help with tweaks and a technical review of this chapter. ❧

103. http://sindresorhus.com/

JavaScript Events And Responding To The User

BY CHRISTIAN HEILMANN ❧

Whenever people ask me about the most powerful things in JavaScript and the DOM, I quickly arrive at events. The reason is that events in browsers are incredibly useful. Furthermore, decoupling functionality from events is a powerful idea, which is why Node.js[104] became such a hot topic.

Today, let's get back to the basics of events and get you in the mood to start playing with them, beyond applying click handlers to everything or breaking the Web with `<a href="javascript:void(0)">` links or messing up our HTML with `onclick="foo()"` inline handlers (I explained in detail in 2005 why these are bad ideas[105]).

Note: This chapter uses plain JavaScript and not any libraries. A lot of what we'll talk about here is easier to achieve in jQuery, YUI or Dojo, but understanding the basics is important because you will find yourself in situations where you cannot use a library but should still be able to deliver an amazing solution.

Disclaimer: The event syntax we'll be using here is `addEventListener()`[106], as defined in the "DOM Level 3 Events[107]" specification, which works in all browsers in use now except for Internet Explorer below version 9. A lot of the things we'll show can be achieved with jQuery, though, which also supports legacy browsers. Come to think of it, one simple `addEventListener()` on `DOMContentLoaded` is a great way to make sure your script does not run on legacy browsers. This is a good thing. If we want the Web to evolve, we need to stop giving complex and demanding code to old browsers. If you build your solutions the right way, then IE 6 would not need any JavaScript to display a workable, albeit simpler, solution. Think of your product as an escalator: if your JavaScript does not execute, the website should still be usable as stairs.

104. http://nodejs.org/
105. http://www.onlinetools.org/articles/unobtrusivejavascript/chapter4.html
106. https://developer.mozilla.org/en/DOM/element.addEventListener
107. http://dev.w3.org/2006/webapi/DOM-Level-3-Events/html/DOM3-Events.html#events-EventTarget-addEventListener

Before we get into the details of events and how to use them, check out a few demos that use scroll events in a clever way to achieve pretty sweet results:

- In its search for a designer, Wealthfront Engineering uses scrolling and shifting content along the Z axis[108]. This was a big part of the Beercamp 2011 website[109]. Wealthfront blogged in detail[110] about how it achieved this.
- Stroll.js[111] takes a slightly similar approach, showing how lovely transitions can be when the user scrolls a list.
- jQuery Scroll Path[112] is a plugin to move content along a path when the user scrolls the page.

All of this is based on event handling and reading out what the browser gives us. Now, let's look at repeating the basics of that.

Basics: What Is An Event?

```
var log = document.getElementById('log'),
    i = '',
    out = [];
for (i in window) {
  if ( /^on/.test(i)) { out[out.length] = i; }
}
log.innerHTML = out.join(', ');
```

In my case, running Firefox, I get this:

```
onmouseenter, onmouseleave, onafterprint, onbeforeprint,
onbeforeunload, onhashchange, onmessage, onoffline,
ononline, onpopstate, onpagehide, onpageshow, onresize,
onunload, ondevicemotion, ondeviceorientation, onabort,
onblur, oncanplay, oncanplaythrough, onchange, onclick,
oncontextmenu, ondblclick, ondrag, ondragend, ondragenter,
ondragleave, ondragover, ondragstart, ondrop,
ondurationchange, onemptied, onended, onerror, onfocus,
```

108. https://www.wealthfront.com/designerwanted
109. http://2011.beercamp.com/
110. http://eng.wealthfront.com/2012/03/scrolling-z-axis-with-css-3d-transforms.html
111. http://lab.hakim.se/scroll-effects/
112. http://joelb.me/scrollpath/

```
oninput, oninvalid, onkeydown, onkeypress, onkeyup, onload,
onloadeddata, onloadedmetadata, onloadstart, onmousedown,
onmousemove, onmouseout, onmouseover, onmouseup,
onmozfullscreenchange, onmozfullscreenerror, onpause,
onplay, onplaying, onprogress, onratechange, onreset,
onscroll, onseeked, onseeking, onselect, onshow, onstalled,
onsubmit, onsuspend, ontimeupdate, onvolumechange,
onwaiting, oncopy, oncut, onpaste, onbeforescriptexecute,
onafterscriptexecute
```

That is a lot to play with, and the way to do that is by using `addEventListener()`:

```
element.addEventListener(event, handler, useCapture);
```

For example:

```
var a = document.querySelector('a'); // grab the first
link in the document
a.addEventListener('click', ajaxloader, false);
```

The `element` is the element that we apply the handler to; as in, "Hey you, link! Make sure you tell me when something happens to you." The `ajaxloader()` function is the event listener; as in, "Hey you! Just stand there and keep your ears and eyes peeled in case something happens to the link." Setting the `useCapture` to `false` means that we are content to capture the event on bubbling, rather than the capturing phase. This is a long and arduous topic[113], well explained on Dev.Opera[114]. Let's just say that by setting the `useCapture` to `false`, you will be fine in 99.7434% of cases (a rough approximation). The parameter is actually optional in all browsers but Opera.

Now, the event handler function gets an object as a parameter from the event, which is full of awesome properties that we can play with. If you try out my example[115], you'll see what the following code does:

```
var log = document.getElementById('log'),
    out = '';

document.addEventListener('click', logeventinfo, false);
```

113. http://www.w3.org/TR/DOM-Level-3-Events/#event-flow
114. http://dev.opera.com/articles/view/event-capture-explained/
115. http://thewebrocks.com/demos/smashing-events/eventproperties.html

```
document.addEventListener('keypress', logeventinfo,
false);

function logeventinfo (ev) {
  log.innerHTML = '';
  out = '<ul>';
  for (var i in ev) {
    if (typeof ev[i] === 'function' || i ===
i.toUpperCase()) {
      continue;
    }
    out += '<li><span>'+i+'</span>: '+ev[i]+'</li>';
  }
  log.innerHTML += out + '</ul>';
}
```

You can assign several event handlers to the same event, or the same handler to various events (as shown in this demo).

The ev is what we get back from the event. And (again, in my case, in Firefox) a lot of interesting things are in it:

```
originalTarget: [object HTMLHtmlElement]
type: click
target: [object HTMLHtmlElement]
currentTarget: [object HTMLDocument]
eventPhase: 3
bubbles: true
cancelable: true
timeStamp: 574553210
defaultPrevented: false
which: 1
rangeParent: [object Text]
rangeOffset: 23
pageX: 182
pageY: 111
isChar: false
screenX: 1016
screenY: 572
clientX: 182
clientY: 111
ctrlKey: false
shiftKey: false
```

```
altKey: false
metaKey: false
button: 0
relatedTarget: null
mozPressure: 0
mozInputSource: 1
view: [object Window]
detail: 1
layerX: 182
layerY: 111
cancelBubble: false
explicitOriginalTarget: [object HTMLHtmlElement]
isTrusted: true
originalTarget: [object HTMLHeadingElement]
type: click
target: [object HTMLHeadingElement]
currentTarget: [object HTMLDocument]
eventPhase: 3
bubbles: true
cancelable: true
timeStamp: 574554192
defaultPrevented: false
which: 1
rangeParent: [object Text]
rangeOffset: 0
pageX: 1
pageY: 18
isChar: false
screenX: 835
screenY: 479
clientX: 1
clientY: 18
ctrlKey: false
shiftKey: false
altKey: false
metaKey: false
button: 0
relatedTarget: null
mozPressure: 0
mozInputSource: 1
view: [object Window]
```

```
detail: 1
layerX: 1
layerY: 18
cancelBubble: false
explicitOriginalTarget: [object Text]
isTrusted: true
```

It also differs from event to event. Try clicking the demo and pressing keys, and you will see that you get different results. You can also refer to the full list of standard `event` properties[116].

The Last Of The Basics: Preventing Execution And Getting The Target

Two more things are important when it comes to events in the browser: we have to stop the browser from carrying out its default action for the event, and we have to find out which element the event fired on. The former is achieved with the `ev.preventDefault()` method, and the latter is stored in `ev.target`.

Say you want to know that a link has been clicked, but you don't want the browser to follow it because you have a great idea of what to do with that event instead. You can do this by subscribing to the click event of the link, and you can stop the browser from following it by calling `preventDefault()`. Here is the HTML:

```
<a class="prevent"
href="http://smashingmagazine.com">Smashing, my dear!</a>
<a class="normal"
href="http://smashingmagazine.com">Smashing, my dear!</a>
```

And the JavaScript:

```
var normal = document.querySelector('.normal'),
    prevent = document.querySelector('.prevent');

prevent.addEventListener('click', function(ev) {
  alert('fabulous, really!');
  ev.preventDefault();
}, false);
```

116. https://developer.mozilla.org/en/DOM/event#Properties

```
normal.addEventListener('click', function(ev) {
  alert('fabulous, really!');
}, false);
```

Note: `document.querySelector()` is the standard way to get an element in the DOM. It is what the `$()` method in jQuery does. You can read the W3C's specification[117] for it and get some explanatory code snippets[118] on the Mozilla Developer Network (MDN).

If you now click the link, you will get an alert. And when you hit the "OK" button, nothing more happens; the browser does not go to `http://smashingmagazine.com`. Without the `preventDefault()`, the browser will show the alert and follow the link. Try it out.[119]

The normal way to access the element that was clicked or hovered over or that had a key pressed is to use the `this` keyword in the handler. This is short and sweet, but it's actually limiting because `addEventListener()` gives us something better: the event target. It could also be confusing because `this` might already be bound to something else, so using `ev.currentTarget` as noted in the specification is a safer bet.

Event Delegation: It Rocks. Use It!

Using the `target` property of the event object, you can find out which element the event occurred on.

Events happen by going down the whole document tree to the element that you interacted with and back up to the main window. This means that if you add an event handler to an element, you will get all of the child elements for free. All you need to do is test the event target and respond accordingly. See my example of a list[120]:

```
<ul id="resources">
  <li><a href="http://developer.mozilla.org">MDN</a></li>
  <li><a href="http://html5doctor.com">HTML5
Doctor</a></li>
  <li><a href="http://html5rocks.com">HTML5
Rocks</a></li>
```

117. http://www.w3.org/TR/selectors-api/
118. https://developer.mozilla.org/En/Code_snippets/QuerySelector
119. http://thewebrocks.com/demos/smashing-events/preventdefault.html
120. http://thewebrocks.com/demos/smashing-events/eventdelegation.html

```
  <li><a
href="http://beta.theexpressiveweb.com/">Expressive
Web</a></li>
  <li><a
href="http://creativeJS.com/">CreativeJS</a></li>
</ul>
```

Hover your mouse over the list in this example and you will see that one event handler is enough to get the links, the list item and the list itself. All you need to do is compare the `tagName` of the event target to what you want to have.

```
var resources = document.querySelector('#resources'),
    log = document.querySelector('#log');

resources.addEventListener('mouseover', showtarget,
false);

function showtarget(ev) {
  var target = ev.target;
  if (target.tagName === 'A') {
    log.innerHTML = 'A link, with the href:' +
target.href;
  }
  if (target.tagName === 'LI') {
    log.innerHTML = 'A list item';
  }
  if (target.tagName === 'UL') {
    log.innerHTML = 'The list itself';
  }
}
```

This means you can save a lot of event handlers—each of which is expensive to the browser. Instead of applying an event handler to each link and responding that way—as most people would do in jQuery with `$('a').click(...)` (although jQuery's `on` is OK)—you can assign a single event handler to the list itself and check which element was just clicked.

The main benefit of this is that you are independent of the HTML. If you add more links at a later stage, there is no need to assign new handlers; the event handler will know automatically that there is a new link to do things with.

Events For Detection, CSS Transitions For Smoothness

If you remember the list of properties earlier in this chapter, there is a lot of things we can use. In the past, we used events for simple hover effects, which now have been replaced with effects using the `:hover` and `:focus` CSS selectors. Some things, however, cannot be done with CSS yet; for example, finding the mouse's position. With an event listener, this is pretty simple. First, we define an element to position, like a ball. The HTML:

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

And the CSS:

```
.plot {
  position:absolute;
  background:rgb(175,50,50);
  width: 20px;
  height: 20px;
  border-radius: 20px;
  display: block;
  top:0;
  left:0;
}
```

We then assign a click handler to the document and position the ball at `PageX`[121] and `pageY`. Notice that we need to subtract half the width of the ball in order to center it on the mouse pointer:

```
var plot = document.querySelector('.plot'),
    offset = plot.offsetWidth / 2;

document.addEventListener('click', function(ev) {
  plot.style.left = (ev.pageX - offset) + 'px';
  plot.style.top = (ev.pageY - offset) + 'px';
}, false);
```

Clicking anywhere on the screen will now move the ball there. However, it's not smooth. If you enable the checkbox in the demo[122], you will

121. https://developer.mozilla.org/en/DOM/event.pageX
122. http://thewebrocks.com/demos/smashing-events/mouseposition.html

see that the ball moves smoothly. We could animate this with a library, but browsers can do better these days. All we need to do is add a transition to the CSS, and then the browser will move the ball smoothly from one position to another. To achieve this, we define a new class named `smooth` and apply it to the plot when the checkbox in the document is clicked. The CSS:

```
.smooth {
  -webkit-transition: 0.5s;
     -moz-transition: 0.5s;
      -ms-transition: 0.5s;
       -o-transition: 0.5s;
          transition: 0.5s;
}
```

The JavaScript:

```
var cb = document.querySelector('input[type=checkbox]');
cb.addEventListener('click', function(ev) {
   plot.classList.toggle('smooth');
}, false);
```

The interplay between CSS and JavaScript events has always been powerful, but it got even better in newer browsers. As you might have guessed, CSS transitions and animations have their own events.

How Long Was A Key Pressed?

As you might have seen in the list of available events earlier, browsers also give us a chance to respond to keyboard entry and tell us when the user has pressed a key. Sadly, though, key handling in a browser is hard to do properly, as Jan Wolter explains in detail[123]. However, as a simple example, let's look how we can measure in milliseconds how long a user has pressed a button. See this keytime demo[124] for an example. Press a key, and you will see the output field grow while the key is down. Once you release the key, you'll see the number of milliseconds that you pressed it. The code is not hard at all:

123. http://unixpapa.com/js/key.html
124. http://thewebrocks.com/demos/smashing-events/keytime.html

```
var resources = document.querySelector('#resources'),
    log = document.querySelector('#log'),
    time = 0;

document.addEventListener('keydown', keydown, false);
document.addEventListener('keyup', keyup, false);

function keydown(ev) {
  if (time === 0) {
    time = ev.timeStamp;
    log.classList.add('animate');
  }
}
function keyup(ev) {
  if (time !== 0) {
    log.innerHTML = ev.timeStamp - time;
    time = 0;
    log.classList.remove('animate');
  }
}
```

We define the elements we want and set the `time` to `0`. We then apply two event handlers to the document, one on `keydown` and one on `keyup`.

In the `keydown` handler, we check whether `time` is `0`, and if it is, we set `time` to the `timeStamp` of the event. We assign a CSS class to the output element, which starts a CSS animation (see the CSS for how that is done).

The `keyup` handler checks whether `time` is still `0` (as `keydown` gets fired continuously while the key is pressed), and it calculates the difference in the time stamps if it isn't. We set `time` back to `0` and remove the class to stop the animation.

Working With CSS Transitions (And Animations)

CSS transitions[125] fire a single event that you can listen for in JavaScript called `transitionend`. The event object then has two properties: `propertyName`, which contains the property that was transitioned, and `elapsedTime`, which tells you how long it took.

125. https://developer.mozilla.org/en/CSS/CSS_transitions

Check out the demo[126] to see it in action. The code is simple enough. Here is the CSS:

```
.plot {
  background:rgb(175,50,50);
  width: 20px;
  height: 20px;
  border-radius: 20px;
  display: block;
  -webkit-transition: 0.5s;
     -moz-transition: 0.5s;
      -ms-transition: 0.5s;
       -o-transition: 0.5s;
          transition: 0.5s;
}

.plot:hover {
  width: 50px;
  height: 50px;
  border-radius: 100px;
  background: blue;
}
```

And the JavaScript:

```
plot.addEventListener('transitionend', function(ev) {
    log.innerHTML += ev.propertyName + ':' +
ev.elapsedTime + 's ';
}, false);
```

This, however, works only in Firefox right now because Chrome, Safari and Opera have vendor-prefixed events instead. As David Calhoun's gist[127] shows, you need to detect what the browser supports and define the event's name that way.

CSS animation events[128] work the same way, but you have three events instead of one: `animationstart`, `animationend` and `animationiteration`. MDN has a demo[129] of it.

126. http://thewebrocks.com/demos/smashing-events/transitionevent.html
127. https://gist.github.com/702826
128. https://developer.mozilla.org/en/CSS/CSS_animations
129. https://developer.mozilla.org/samples/cssref/animations/animevents.html

Speed, Distance And Angle

Detecting events happening is one thing. If you want to do something with them that is a beautiful and engaging, then you need to go further and put some math into it. So, let's have a go at using a few mouse handlers to calculate the angle, distance and speed of movement when a user drags an element across the screen. Check out the demo first.[130]

```
var plot = document.querySelector('.plot'),
    log = document.querySelector('output'),
    offset = plot.offsetWidth / 2,
    pressed = false,
    start = 0, x = 0, y = 0, end = 0, ex = 0, ey = 0, mx
= 0, my = 0,
    duration = 0, dist = 0, angle = 0;

document.addEventListener('mousedown', onmousedown,
false);
document.addEventListener('mouseup', onmouseup, false);
document.addEventListener('mousemove', onmousemove,
false);

function onmousedown(ev) {
  if (start === 0 && x === 0 && y === 0) {
    start = ev.timeStamp;
    x = ev.clientX;
    y = ev.clientY;
    moveplot(x, y);
    pressed = true;
  }
}
function onmouseup(ev) {
  end = ev.timeStamp;
  duration = end - start;
  ex = ev.clientX;
  ey = ev.clientY;
  mx = ex - x;
  my = ey - y;
  dist = Math.sqrt(mx * mx + my * my);
  start = x = y = 0;
```

130. http://thewebrocks.com/demos/smashing-events/speeddistanceangle.html

```
  pressed = false;
  angle = Math.atan2( my, mx ) * 180 / Math.PI;
  log.innerHTML = '<strong>' + (dist>>0) +'</strong>
pixels in <strong>'+
                  duration +'</strong> ms ( <strong>' +
                  twofloat(dist/duration) +'</strong>
pixels/ms)'+
                  ' at <strong>' + twofloat(angle) +
                  '</strong> degrees';
}
function onmousemove (ev) {
  if (pressed) {
    moveplot(ev.pageX, ev.pageY);
  }
}
function twofloat(val) {
  return Math.round((val*100))/100;
}
function moveplot(x, y) {
  plot.style.left = (x - offset) + 'px';
  plot.style.top = (y - offset) + 'px';
}
```

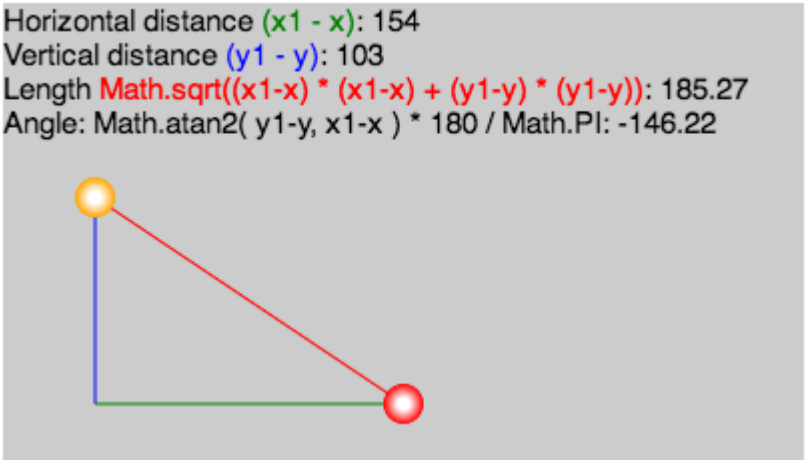
OK, I admit: quite a lot is going on here. But it is not as hard as it looks. For both `onmousedown` and `onmouseup`, we read the mouse's position with `clientX` and `clientY` and the `timeStamp` of the event. Mouse events have time stamps that tell you when they happened. When the mouse moves, all we check is whether the mouse button has been pressed (via a boolean set in the `mousedown` handler) and move the plot with the mouse.

The rest is geometry—good old Pythagoras[131], to be precise. We get the speed of the movement by checking the number of pixels traveled in the time difference between `mousedown` and `mouseup`.

We get the number of pixels traveled as the square root of the sum of the squares of the difference between x and y at the start and end of the movement. And we get the angle by calculating the arctangent of the triangle. All of this is covered in "A Quick Look Into the Math of Ani-

131. http://en.wikipedia.org/wiki/Pythagorean_theorem

mations With JavaScript[132]"; or you can play with the following JSFiddle example:



Demo on JS Fiddle[133].

Media Events

Both video and audio fire a lot of events that we can tap into. The most interesting are the time events that tell you how long a song or movie has been playing. A nice little demo to look at is the MGM-inspired dinosaur animation[134] on MDN; I recorded a six-minute screencast[135] explaining how it is done.

If you want to see a demo of all the events in action, the JPlayer team has a great demo page showing media events[136].

Input Options

Traditionally, browsers gave us mouse and keyboard interaction. Nowadays, this is not enough because we use hardware that offers more to us. Device orientation[137], for example, allows you to respond to the tilting of a phone or tablet; touch events[138] are a big thing on mobiles and tablets; the Gamepad API[139] allows us to read out game controllers in browsers; postMessage[140] allows us to send messages across

132. http://coding.smashingmagazine.com/2011/10/04/quick-look-math-animations-javascript/
133. http://jsfiddle.net/codepo8/bAwUf/embedded/result,js,html,css
134. http://hacks.mozilla.org/2012/03/making-the-dino-roar-syncing-audio-and-css-transitions/
135. http://www.youtube.com/watch?v=aFIJ_ZpV-8Q
136. http://www.jplayer.org/HTML5.Media.Event.Inspector/
137. https://developer.mozilla.org/en/DOM/DeviceOrientationEvent
138. https://developer.mozilla.org/en/DOM/Touch_events

domains and browser windows; pageVisibility[141] allows us to react to users switching to another tab. We can even detect when the history object of the window has been manipulated[142]. Check the list of events in the window object to find some more gems that might not be quite ready but should be available soon for us to dig into.

Whatever comes next in browser support, you can be sure that events will be fired and that you will be able to listen to them. The method works and actually rocks.

Go Out And Play

And that is it. Events are not hard; in most cases, you just need to subscribe to them and check what comes back as the event object to see what you can do with it. Of course, a lot of browser hacking is still needed at times, but I for one find incredible the number of ways we can interact with our users and see what they are doing. If you want to get really creative with this, stop thinking about the use cases we have now and get down into the nitty gritty of what distances, angles, speed and input can mean to an interface. If you think about it, playing Angry Birds to the largest degree means detecting the start and end of a touch event and detecting the power and direction that the bird should take off in. So, what is stopping you from creating something very interactive and cool? ❧

139. http://hacks.mozilla.org/2011/12/paving-the-way-for-open-games-on-the-web-with-the-gamepad-and-mouse-lock-apis/
140. https://developer.mozilla.org/en/DOM/window.postMessage
141. https://developer.mozilla.org/en/DOM/Using_the_Page_Visibility_API
142. https://developer.mozilla.org/en/DOM/window.onpopstate

JavaScript Profiling With The Chrome Developer Tools

BY ZACK GROSSBART ❧

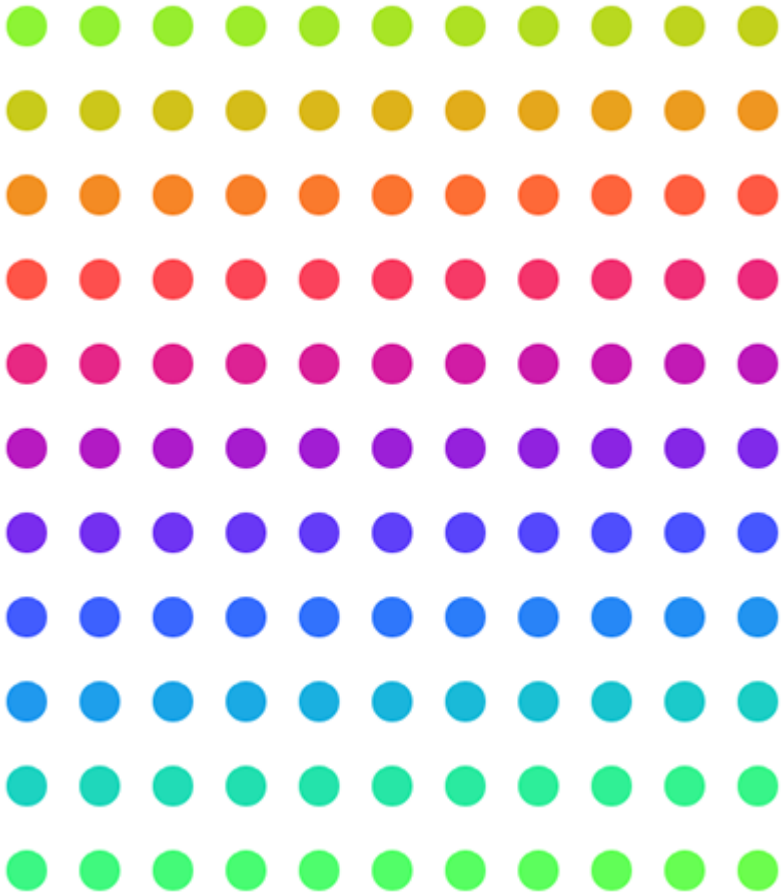
Your website works. Now let's make it work faster. Website performance is about two things: how fast the page loads, and how fast the code on it runs. Plenty of services make your website load faster, from minimizers[143] to CDNs[144], but making it run faster is up to you.

Little changes in your code can have gigantic performance impacts. A few lines here or there could mean the difference between a blazingly fast website and the dreaded "Unresponsive Script" dialog. This chapter shows you a few ways to find those lines of code with Chrome Developer Tools[145].

Establish A Baseline

We'll look at a simple application called a color sorter, which presents a grid of rainbow colors that you can drag and drop to mix up. Each dot is a `div` tag with a little CSS to make it look like a circle.

143. http://stackoverflow.com/questions/3520285/is-there-a-good-javascript-minimizer
144. http://en.wikipedia.org/wiki/Content_delivery_network
145. http://code.google.com/chrome/devtools/docs/overview.html



Generating my rainbow colors was a little tricky, so I got help from "Making Annoying Rainbows in JavaScript[146]."

The page loads pretty fast, but it still takes a moment and blinks a little before it paints. Time to profile the page and make it faster.

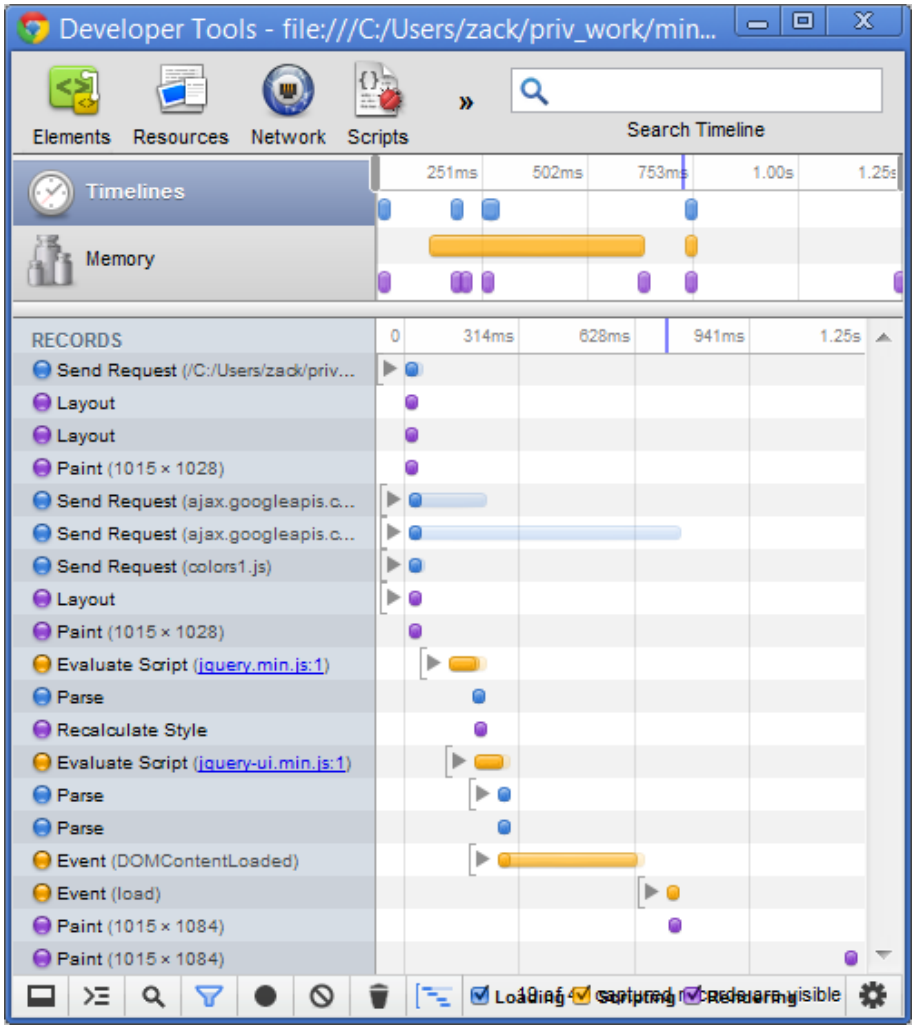
Always start performance-improvement projects with a baseline understanding of how fast or slow your application already is. The baseline will let you know whether you're making improvements and help you make tradeoffs. For this chapter, we'll use Chrome Developer Tools[147].

The profiler is part of Chrome Developer Tools, which is always available in Chrome. Click the "Tools" menu under the little wrench to open it. Firebug[148] has some profiling tools, too, but the WebKit browsers (Chrome and Safari) are best at profiling code and showing timelines. Chrome also offers an excellent tool for event tracing, called Speed Tracer[149].

To establish our baseline, we'll start recording in the "Timeline" tab, load our page and then stop the recording. (To start recording once

146. http://krazydad.com/tutorials/makecolors.php
147. http://code.google.com/chrome/devtools/docs/overview.html
148. http://getfirebug.com
149. https://developers.google.com/web-toolkit/speedtracer/

Chrome Developer Tools is open, click the "Timeline" tab, and then the small black circle icon for "Record" at the very bottom of the window.) Chrome is smart about not starting to record until the page starts to load. I run it three times and take the average, in case my computer runs slowly during the first test.



My average baseline—i.e. the time between the first request for the page and the final painting of the page in the browser—is 1.25 seconds. That's not bad, but it's not great for such a small page.

I want to make my code run faster, but I'm not sure what's making it slow. The profiler helps me find out.

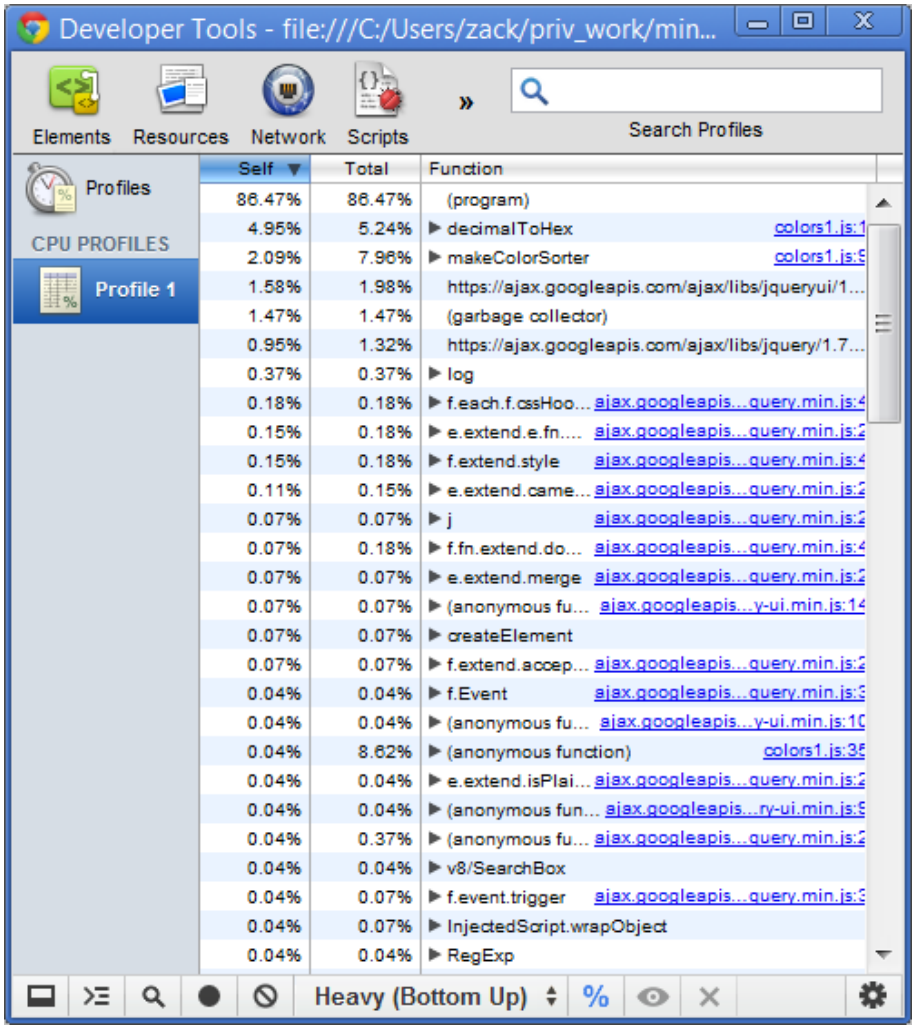
Create A Profile

The timeline tells us how long our code took to run, but that doesn't help us know what's going on while it's running. We could make changes and run the timeline again and again, but that's just shooting in the dark. The "Profiles" tab gives us a better way to see what's going on.

Profilers show us which functions take the most time. Let's make our baseline profile by switching to the "Profiles" tab in Chrome Developer Tools, where three types of profiling are offered:

1. **JavaScript CPU profile**
   Shows how much CPU time our JavaScript is taking.
2. **CSS selector profile**
   Shows how much CPU time is spent processing CSS selectors.
3. **Heap snapshot**
   Shows how memory is being used by our JavaScript objects.

We want to make our JavaScript run faster, so we'll use the CPU profiling. We start the profile, refresh the page and then stop the profiler.



The first thing that's clear from the profile is that a lot is going on. The color sorter uses jQuery and jQuery UI, which are doing a lot of stuff like managing plugins and parsing regular expressions. I can also see that two of my functions are at the top of the list: `decimalToHex` and

`makeColorSorter`. These two functions take a total of 13.2% of my CPU time, so they're a good place to start making improvements.

We can click the arrow next to the function calls to open the complete function-call stack. Expanding them, I see that `decimalToHex` is called from `makeColorSorter`, and `makeColorSorter` is called from `$(document).ready`.

Here's the code:

```
$(document).ready(function() {
    makeColorSorter(.05, .05, .05, 0, 2, 4, 128, 127,
121);
    makeSortable();
});
```

Knowing where they're called from also makes clear that making the colors sortable isn't my biggest performance problem. Performance issues resulting from the addition of a lot of sortables is common[150], but my code is taking more time to add DOM elements than to make them sortable.

I want to start making those functions faster, but first I want to isolate my changes. A lot happens when the page loads, and I want to get all of that out of my profile.

Isolate The Problem

Instead of loading the color sorter when the document is ready, I'll make a second version[151] that waits until I press a button to load the color sorter. This isolates it from the document loading and helps me profile just the code. I can change it back once I'm done tuning performance.

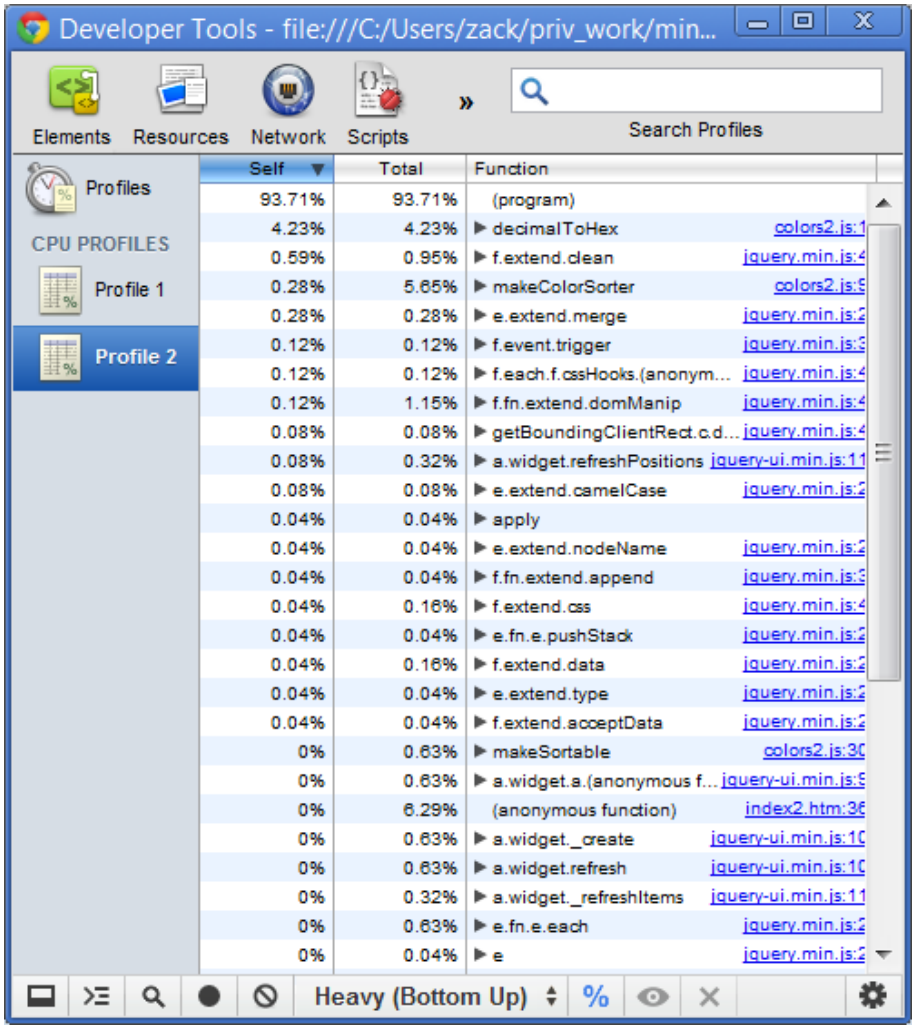
Let's call the new function `testColorSorter` and bind it to a clickable button:

```
function testColorSorter() {
    makeColorSorter(.05, .05, .05, 0, 2, 4, 128, 127,
121);
    makeSortable();
}
```

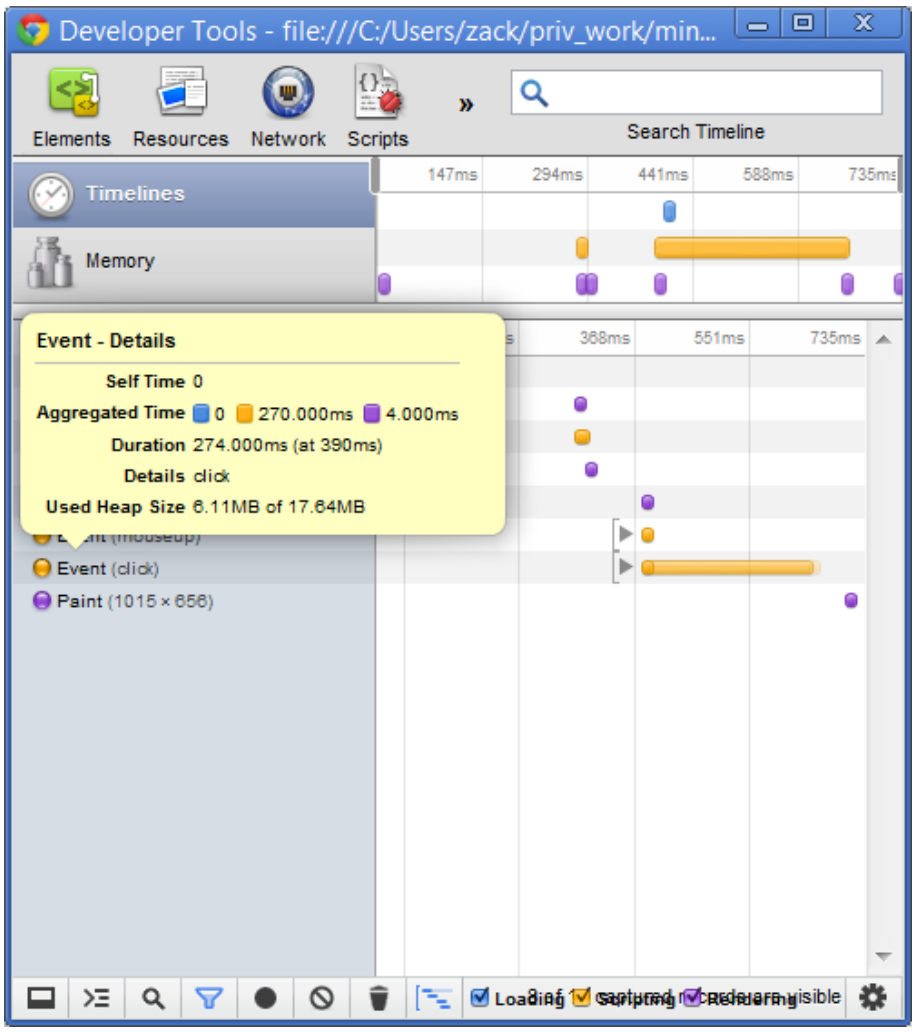
150. http://37signals.com/svn/posts/3137-using-event-capturing-to-improve-basecamp-page-load-times
151. http://zgrossbart.github.com/jsprofarticle/index2.htm

```
<button id="clickMe" onclick="testColorSorter();">Click
me</button>
```

Changing the application before we profile could alter the performance of the application unexpectedly. This change looks pretty safe, but I still want to run the profiler again to see whether I've accidentally changed anything else. I'll create this new profile by starting the profiler, pressing the button in the app and then stopping the profile.



The first thing to notice is that the `decimalToHex` function is now taking up 4.23% of the time to load; it's what the code spends the most time on. Let's create a new baseline to see how much the code improves in this scenario.



A few events occur before I press the button, but I only care about how long it took between the times the mouse was clicked and the browser painted the color sorter. The mouse button was clicked at 390 milliseconds, and the paint event happened at 726 milliseconds; 726 minus 390 equals my baseline of 336 milliseconds. Just as with the first baseline, I ran it three times and took the average time.

At this point, I know where to look and how long the code takes to run. Now we're ready to start fixing the problem.

Make It Faster

The profiler only tells us which function is causing the problem, so we need to look into it and understand what it does.

```
function decimalToHex(d) {
    var hex = Number(d).toString(16);
    hex = "00".substr(0, 2 - hex.length) + hex;
```

```
        console.log('converting ' + d + ' to ' + hex);
        return hex;
    }
```

Each dot in the color sorter takes a background color value in hex format[152], such as `#86F01B` or `#2456FE`. These values represent the red, green and blue values of the color. For example,

has a background color of `#2456FE`, which means a red value of 36, a green value of 86 and a blue value of 254. Each value must be between 0 and 255.

The `decimalToHex` function converts these RGB[153] colors to hex colors that we can use on the page.

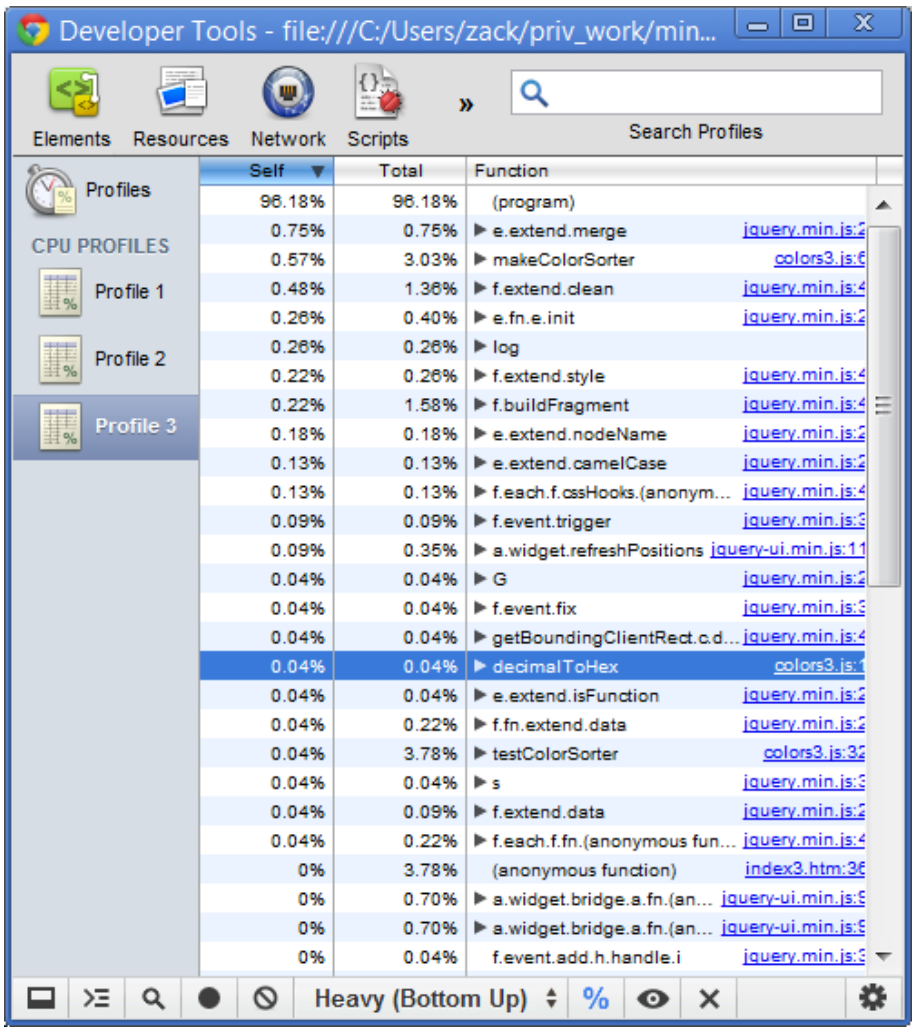
The function is pretty simple, but I've left a `console.log` message in there, which is just some debugging code we can remove.

The `decimalToHex` function also adds padding to the beginning of the number. This is important because some base-10 numbers result in a single hex digit; 12 in base 10 is C in hex, but CSS requires two digits. We can make the conversion faster by making it a little less generic. I know that the numbers to be padded each have one digit, so we can rewrite the function like this:

```
    function decimalToHex(d) {
        var hex = Number(d).toString(16);
        return hex.length === 1 ? '0' + hex : hex; }
```

Version three[154] of the color sorter changes the string only when it needs the padding and doesn't have to call `substr`. With this new function, our runtime is 137 milliseconds. By profiling the code again, I can see that the `decimalToHex` function now takes only 0.04% of the total time—putting it way down the list.

152. http://en.wikipedia.org/wiki/Hexadecimal
153. http://en.wikipedia.org/wiki/Rgb
154. http://zgrossbart.github.com/jsprofarticle/index3.htm



We can also see that the function using the most CPU is `e.extend.merge` from jQuery. I'm not sure what that function does because the code is minimized. I could add the development version of jQuery, but I can see that the function is getting called from `makeColorSorter`, so let's make that one faster next.

Minimize Content Changes

The rainbow colors in the color sorter are generated from a sine wave. The code looks at a center point in the color spectrum and creates a wave through that center point over a specified width. This changes the colors into a rainbow pattern. We can also change the colors in the rainbow by changing the frequency of the red, green and blue.

```
function makeColorSorter(frequency1, frequency2,
frequency3, phase1, phase2, phase3, center, width, len) {

    for (var i = 0; i < len; ++i)
```

```
    {
        var red = Math.floor(Math.sin(frequency1 * i +
phase1) * width + center);
        var green = Math.floor(Math.sin(frequency2 * i +
phase2) * width + center);
        var blue = Math.floor(Math.sin(frequency3 * i +
phase3) * width + center);

        console.log('red: ' + decimalToHex(red));
        console.log('green: ' + decimalToHex(green));
        console.log('blue: ' + decimalToHex(blue));

        var div = $('<div class="colorBlock"></div>');
        div.css('background-color', '#' +
decimalToHex(red) + decimalToHex(green) +
decimalToHex(blue));
        $('#colors').append(div);

    }
}
```

We could take out more `console.log` functions. The calls are especially bad because each is also calling the `decimalToHex` function, which means that `decimalToHex` is effectively being called twice as often as it should.

This function changes the DOM a lot. Every time the loop runs, it adds a new `div` to the `colors` div tag. This makes me wonder whether that's what the `e.extend.merge` function was doing. The profiler makes it easy to tell with a simple experiment.

Instead of adding a new `div` each time the loop runs, I want to add all of the `div` tags at once. Let's create a variable to hold them, and then add them once at the end.

```
function makeColorSorter(frequency1, frequency2,
frequency3,
  phase1, phase2, phase3,
  center, width, len) {

    var colors = "";
    for (var i = 0; i < len; ++i)
    {
        var red = Math.floor(Math.sin(frequency1 * i +
```

```
phase1) * width + center);
        var green = Math.floor(Math.sin(frequency2 * i +
phase2) * width + center);
        var blue = Math.floor(Math.sin(frequency3 * i +
phase3) * width + center);

        colors += '<div class="colorBlock"
style="background-color: #' +
            decimalToHex(red) + decimalToHex(green) +
decimalToHex(blue) + '"></div>';
    }

    $('#colors').append(colors);
}
```

This small change in the code means that the DOM changes once, when it adds all of the `div` tags. Testing that with the timeline, we see that the runtime between the click and the paint events is now 31 milliseconds. This one DOM change has brought the time for version four[155] down by about 87%. We can also run the profiler again and see that the `e.extend.merge` function now takes up such a small percentage of the time that it doesn't show up on the list.

We could make the code one notch faster by removing the `decimalToHex` function entirely. CSS supports RGB colors, so we don't need to convert them to hex. Now we can write our `makeColorSorter` function like this:

```
function makeColorSorter(frequency1, frequency2,
frequency3,
  phase1, phase2, phase3,
  center, width, len) {

    var colors = "";
    for (var i = 0; i < len; ++i)
    {
        var red = Math.floor(Math.sin(frequency1 * i +
phase1) * width + center);
        var green = Math.floor(Math.sin(frequency2 * i +
phase2) * width + center);
```

155. http://zgrossbart.github.com/jsprofarticle/index4.htm

```
        var blue = Math.floor(Math.sin(frequency3 * i +
phase3) * width + center);

        colors += '<div class="colorBlock"
style="background-color: rgb(' +
            red + ',' + green + ',' + blue + ')"></div>';
    }

    $('#colors').append(colors);
}
```

Version five[156] runs in only 26 milliseconds and uses 18 lines of code for what used to take 28 lines.

JavaScript Profiling In Your Application

Real-world applications are much more complex than this color sorter, but profiling them follows the same basic steps:

1. **Establish a baseline** so that you know where you're starting from.
2. **Isolate the problem** from any other code running in the application.
3. **Make it faster** in a controlled environment, with frequent timelines and profiles.

There are a few other rules to follow when tuning performance:

1. **Start with the slowest parts first** so that you get the most improvement for the time spent tuning.
2. **Control the environment.** If you switch computers or make any other major changes, always run a new baseline.
3. **Repeat the analysis** to prevent anomalies on your computer from skewing the results.

Everyone wants their website to run faster. You have to develop new features, but new features usually make a website run slower. So, investing time in tuning the performance does pay off.

156. http://zgrossbart.github.com/jsprofarticle/index5.htm

Profiling and tuning cut the final color sorter[157]'s runtime by over 92%. How much faster could your website be? ❧

157. http://zgrossbart.github.com/jsprofarticle/index6.htm

Writing Fast, Memory-Efficient JavaScript

BY ADDY OSMANI

JavaScript engines such as Google's V8[158] (Chrome, Node) are specifically designed for the fast execution[159] of large JavaScript applications. As you develop, if you care about memory usage and performance, you should be aware of some of what's going on in your user's browser's JavaScript engine behind the scenes.

Whether it's V8, SpiderMonkey[160] (Firefox), Carakan[161] (Opera), Chakra[162] (IE) or something else, doing so can help you **better optimize your applications**. That's not to say one should optimize for a single browser or engine. Never do that. You should, however, ask yourself questions such as:

- Is there anything I could be doing more efficiently in my code?
- What (common) optimizations do popular JavaScript engines make?
- What is the engine unable to optimize for, and is the garbage collector able to clean up what I'm expecting it to?

158. http://code.google.com/p/v8/
159. http://www.html5rocks.com/en/tutorials/speed/v8/
160. https://developer.mozilla.org/en-US/docs/SpiderMonkey
161. http://my.opera.com/ODIN/blog/carakan-faq
162. http://en.wikipedia.org/wiki/Chakra_(JScript_engine)

Fast-loading Web sites—like fast cars—require the use specialized tools. Image source: dHybridcars[163].

There are many common pitfalls when it comes to writing memory-efficient and fast code, and in this chapter we're going to explore some test-proven approaches for writing code that performs better.

So, How Does JavaScript Work In V8?

While it's possible to develop large-scale applications without a thorough understanding of JavaScript engines, any car owner will tell you they've looked under the hood at least once. As Chrome is my browser of choice, I'm going to talk a little about its JavaScript engine. V8 is made up of a few core pieces.

- A **base compiler**, which parses your JavaScript and generates native machine code before it is executed, rather than executing bytecode or simply interpreting it. This code is initially not highly optimized.
- V8 represents your objects in an **object model**. Objects are represented as associative arrays in JavaScript, but in V8 they are represented with hidden classes[164], which are an internal type system for optimized lookups.
- The **runtime profiler** monitors the system being run and identifies "hot" functions (i.e. code that ends up spending a long time running).

163. http://dhybridcars.com/toyota-hybrid/2013-scion-fr-s-sexy-sport-car/media/2013-scion-fr-s-speed-gauge-img-8/
164. https://developers.google.com/v8/design

- An **optimizing compiler** recompiles and optimizes the "hot" code identified by the runtime profiler, and performs optimizations such as inlining (i.e. replacing a function call site with the body of the callee).
- V8 supports **deoptimization**, meaning the optimizing compiler can bail out of code generated if it discovers that some of the assumptions it made about the optimized code were too optimistic.
- It has a **garbage collector**. Understanding how it works can be just as important as the optimized JavaScript.

Garbage Collection

Garbage collection is **a form of memory management**. It's where we have the notion of a collector which attempts to reclaim memory occupied by objects that are no longer being used. In a garbage-collected language such as JavaScript, objects that are still referenced by your application are not cleaned up.

Manually de-referencing objects is not necessary in most cases. By simply putting the variables where they need to be (ideally, as local as possible, i.e. inside the function where they are used versus an outer scope), things should just work.

Garbage collection attempts to reclaim memory. Image source: Valtteri Mäki[165].

It's not possible to force garbage collection in JavaScript. You wouldn't want to do this, because the garbage collection process is controlled by the runtime, and it generally knows best when things should be cleaned up.

DE-REFERENCING MISCONCEPTIONS

In quite a few discussions online about reclaiming memory in JavaScript, the `delete` keyword is brought up, as although it was supposed to be used for just removing keys from a map, some developers think you can force de-referencing using it. Avoid using `delete` if you can. In the below example, `delete o.x` does a lot more harm than good behind the scenes, as it changes `o`'s hidden class and makes it a generic slow object.

```
var o = { x: 1 };
delete o.x; // true
o.x; // undefined
```

That said, you are almost certain to find references to `delete` in many popular JavaScript libraries – it does have a purpose in the language. The main takeaway here is to avoid modifying the structure of hot objects at runtime. JavaScript engines can detect such "hot" objects and attempt to optimize them. This is easier if the object's structure doesn't heavily change over its lifetime and `delete` can trigger such changes.

There are also misconceptions about how `null` works. Setting an object reference to `null` doesn't "null" the object. It sets the object reference to `null`. Using `o.x = null` is better than using `delete`, but it's probably not even necessary.

```
var o = { x: 1 };
o = null;
o; // null
o.x // TypeError
```

If this reference was the last reference to the object, the object is then eligible for garbage collection. If the reference was not the last reference to the object, the object is reachable and will not be garbage collected.

Another important note to be aware of is that global variables are not cleaned up by the garbage collector during the life of your page. Re-

165. http://www.flickr.com/photos/26817893@N05/2864644153/

gardless of how long the page is open, variables scoped to the JavaScript runtime global object will stick around.

```
var myGlobalNamespace = {};
```

Globals are cleaned up when you refresh the page, navigate to a different page, close tabs or exit your browser. Function-scoped variables get cleaned up when a variable falls out of scope. When functions have exited and there aren't any more references to it, the variable gets cleaned up.

RULES OF THUMB

To give the garbage collector a chance to collect as many objects as possible as early as possible, **don't hold on to objects you no longer need**. This mostly happens automatically; here are a few things to keep in mind.

- As mentioned earlier, a better alternative to manual de-referencing is to use variables with an appropriate scope. I.e. instead of a global variable that's nulled out, just use a function-local variable that goes out of scope when it's no longer needed. This means cleaner code with less to worry about.
- Ensure that you're unbinding event listeners where they are no longer required, especially when the DOM objects they're bound to are about to be removed
- If you're using a data cache locally, make sure to clean that cache or use an aging mechanism to avoid large chunks of data being stored that you're unlikely to reuse

FUNCTIONS

Next, let's look at functions. As we've already said, garbage collection works by reclaiming blocks of memory (objects) which are no longer reachable. To better illustrate this, here are some examples.

```
function foo() {
    var bar = new LargeObject();
    bar.someCall();
}
```

When `foo` returns, the object which `bar` points to is automatically available for garbage collection, because there is nothing left that has a reference to it. Compare this to:

```
function foo() {
    var bar = new LargeObject();
    bar.someCall();
    return bar;
}

// somewhere else
var b = foo();
```

We now have a reference to the object which survives the call and persists until the caller assigns something else to `b` (or `b` goes out of scope).

CLOSURES

When you see a function that returns an inner function, that inner function will have access to the outer scope even after the outer function is executed. This is basically a closure[166]—an expression which can work with variables set within a specific context. For example:

```
function sum (x) {
    function sumIt(y) {
        return x + y;
    };
    return sumIt;
}

// Usage
var sumA = sum(4);
var sumB = sumA(3);
console.log(sumB); // Returns 7
```

The function object created within the execution context of the call to `sum` can't be garbage collected, as it's referenced by a global variable and is still very much accessible. It can still be executed via `sumA(n)`.

Let's look at another example. Here, can we access `largeStr`?

166. http://robertnyman.com/2008/10/09/explaining-javascript-scope-and-closures/

```
var a = function () {
    var largeStr = new Array(1000000).join('x');
    return function () {
        return largeStr;
    };
}();
```

Yes, we can, via `a()`, so it's not collected. How about this one?

```
var a = function () {
    var smallStr = 'x';
    var largeStr = new Array(1000000).join('x');
    return function (n) {
        return smallStr;
    };
}();
```

We can't access it anymore and it's a candidate for garbage collection.

TIMERS

One of the worst places to leak is in a loop, or in `setTimeout()`/`setInterval()`, but this is quite common. Consider the following example.

```
var myObj = {
    callMeMaybe: function () {
        var myRef = this;
        var val = setTimeout(function () {
            console.log('Time is running out!');
            myRef.callMeMaybe();
        }, 1000);
    }
};
```

If we then run:

```
myObj.callMeMaybe();
```

to begin the timer, we can see every second "Time is running out!" If we then run:

```
myObj = null;
```

The timer will still fire. `myObj` won't be garbage collected as the closure passed to `setTimeout` has to be kept alive in order to be executed. In turn, it holds references to `myObj` as it captures `myRef`. This would be the same if we'd passed the closure to any other function, keeping references to it.

It is also worth keeping in mind that references inside a `setTimeout`/`setInterval` call, such as functions, will need to execute and complete before they can be garbage collected.

Be Aware Of Performance Traps

It's important never to optimize code until you actually need to. This can't be stressed enough. It's easy to see a number of micro-benchmarks showing that N is more optimal than M in V8, but test it in a real module of code or in an actual application, and **the true impact of those optimizations may be much more minimal** than you were expecting.

Doing too much can be as harmful as not doing anything. Image source: Tim Sheerman-Chase[167].

Let's say we want to create a module which:

- Takes a local source of data containing items with a numeric ID,
- Draws a table containing this data,

167. http://www.flickr.com/photos/tim_uk/7717078488/sizes/c/in/photostream/

- Adds event handlers for toggling a class when a user clicks on any cell.

There are a few different factors to this problem, even though it's quite straightforward to solve. How do we store the data? How do we efficiently draw the table and append it to the DOM? How do we handle events on this table optimally?

A first (naive) take on this problem might be to store each piece of available data in an object which we group into an array. One might use jQuery to iterate through the data and draw the table, then append it to the DOM. Finally, one might use event binding for adding the click behavior we desire.

Note: This is NOT what you should be doing

```
var moduleA = function () {
    return {
        data: dataArrayObject,
        init: function () {
            this.addTable();
            this.addEvents();
        },

        addTable: function () {
            for (var i = 0; i < rows; i++) {
                $tr = $('<tr></tr>');
                for (var j = 0; j < this.data.length;
j++) {
                    $tr.append('<td>' +
this.data[j]['id'] + '</td>');
                }
                $tr.appendTo($tbody);
            }
        },
        addEvents: function () {
            $('table td').on('click', function () {
                $(this).toggleClass('active');
            });
        }
    };
}();
```

Simple, but it gets the job done. In this case however, the only data we're iterating are IDs, a numeric property which could be more simply represented in a standard array. Interestingly, directly using `Document-`

`Fragment` and native DOM methods are more optimal than using jQuery (in this manner) for our table generation, and of course, event delegation is typically more performant than binding each `td` individually.

Note that jQuery does use `DocumentFragment` internally behind the scenes, but in our example, the code is calling `append()` within a loop and each of these calls has little knowledge of the other so it may not be able to optimize for this example. This should hopefully not be a pain point, but be sure to benchmark your own code to be sure.

In our case, adding in these changes results in some good (expected) performance gains. Event delegation provides decent improvement over simply binding, and opting for `documentFragment`[168] was a real booster.

```
var moduleD = function () {
    return {
        data: dataArray,
        init: function () {
            this.addTable();
            this.addEvents();
        },
        addTable: function () {
            var td, tr;
            var frag = document.createDocumentFragment();
            var frag2 =
document.createDocumentFragment();

            for (var i = 0; i < rows; i++) {
                tr = document.createElement('tr');
                for (var j = 0; j < this.data.length;
j++) {
                    td = document.createElement('td');

td.appendChild(document.createTextNode(this.data[j]));

                    frag2.appendChild(td);
                }
                tr.appendChild(frag2);
                frag.appendChild(tr);
            }
```

168. http://jsperf.com/first-pass

```
            tbody.appendChild(frag);
        },
        addEvents: function () {
            $('table').on('click', 'td', function () {
                $(this).toggleClass('active');
            });
        }
    };
}();
```

We might then look to other ways of improving performance. You may have read somewhere that using the prototypal pattern is more optimal than the module pattern (we confirmed it wasn't earlier), or heard that using JavaScript templating frameworks are highly optimized. Sometimes they are, but use them because they make for readable code. Also, precompile!. Let's test and find out how true this hold in practice.

```
moduleG = function () {};

moduleG.prototype.data = dataArray;
moduleG.prototype.init = function () {
    this.addTable();
    this.addEvents();
};
moduleG.prototype.addTable = function () {
    var template = _.template($('#template').text());
    var html = template({'data' : this.data});
    $tbody.append(html);
};
moduleG.prototype.addEvents = function () {
   $('table').on('click', 'td', function () {
       $(this).toggleClass('active');
   });
};

var modG = new moduleG();
```

As it turns out, in this case the performance benefits are negligible. Opting for templating and prototypes[169] didn't really offer anything more than what we had before. That said, performance isn't really the reason

169. http://jsperf.com/second-pass

modern developers use either of these things—it's the readability, inheritance model and maintainability they bring to your codebase.

More complex problems include efficiently drawing images using canvas[170] and manipulating pixel data[171] with or without typed arrays[172]

Always give micro-benchmarks a close lookover before exploring their use in your application. Some of you may recall the JavaScript templating shoot-off[173] and the extended shoot-off that followed[174]. You want to make sure that tests aren't being impacted by constraints you're unlikely to see in real world applications—test optimizations together in actual code.

V8 Optimization Tips

Whilst detailing every V8 optimization is outside the scope of this chapter, there are certainly many tips worth noting. Keep these in mind and you'll reduce your chances of writing unperformant code.

- Certain patterns will cause V8 to bail out of optimizations. A try-catch, for example, will cause such a bailout. For more information on what functions can and can't be optimized, you can use `--trace-opt file.js` with the d8 shell utility that comes with V8.
- If you care about speed, try very hard to keep your functions monomorphic, i.e. make sure that variables (including properties, arrays and function parameters) only ever contain objects with the same hidden class. For example, don't do this:

```
function add(x, y) {
   return x+y;
}

add(1, 2);
add('a','b');
add(my_custom_object, undefined);
```

170. http://jsperf.com/canvas-drawimage-vs-webgl-drawarrays/6
171. http://jsperf.com/canvas-pixel-manipulation/30
172. http://jsperf.com/typed-arrays-for-pixel-manipulation
173. http://jsperf.com/dom-vs-innerhtml-based-templating/473
174. http://jsperf.com/javascript-templating-shootoff-extended/26

- Don't load from uninitialized or deleted elements. This won't make a difference in output, but it will make things slower.
- Don't write enormous functions, as they are more difficult to optimize

For more tips, watch Daniel Clifford's Google I/O talk Breaking the JavaScript Speed Limit with V8[175] as it covers these topics well. Optimizing For V8—A Series[176] is also worth a read.

OBJECTS VS. ARRAYS: WHICH SHOULD I USE?

- If you want to store a bunch of numbers, or a list of objects of the same type, use an array.
- If what you semantically need is an object with a bunch of properties (of varying types), use an object with properties. That's pretty efficient in terms of memory, and it's also pretty fast.
- Integer-indexed elements, regardless of whether they're stored in an array or an object, are much faster to iterate over than object properties[177].
- Properties on objects are quite complex: they can be created with setters, and with differing enumerability and writability. Items in arrays aren't able to be customized as heavily—they either exist or they don't. At an engine level, this allows for more optimization in terms of organizing the memory representing the structure. This is particularly beneficial when the array contains numbers. For example, when you need vectors, don't define a class with properties x, y, z; use an array instead..

There's really only one major difference between objects and arrays in JavaScript, and that's the arrays' magic `length` property. If you're keeping track of this property yourself, objects in V8 should be just as fast as arrays.

TIPS WHEN USING OBJECTS

- Create objects using a constructor function. This ensures that all objects created with it have the same hidden class and helps avoid chang-

175. http://www.youtube.com/watch?v=UJPdhx5zTaw
176. http://floitsch.blogspot.co.uk/2012/03/optimizing-for-v8-introduction.html
177. http://jsperf.com/performance-of-array-vs-object/3

ing these classes. As an added benefit, it's also slightly faster than `Object.create()`[178]

- There are no restrictions on the number of different object types you can use in your application or on their complexity (within reason: long prototype chains tend to hurt, and objects with only a handful of properties get a special representation that's a bit faster than bigger objects). For "hot" objects, try to keep the prototype chains short and the field count low.

Object Cloning

Object cloning is a common problem for app developers. While it's possible to benchmark how well various implementations work with this type of problem in V8, be very careful when copying anything. Copying big things is generally slow—don't do it. `for..in` loops in JavaScript are particularly bad for this, as they have a devilish specification and will likely never be fast in any engine for arbitrary objects.

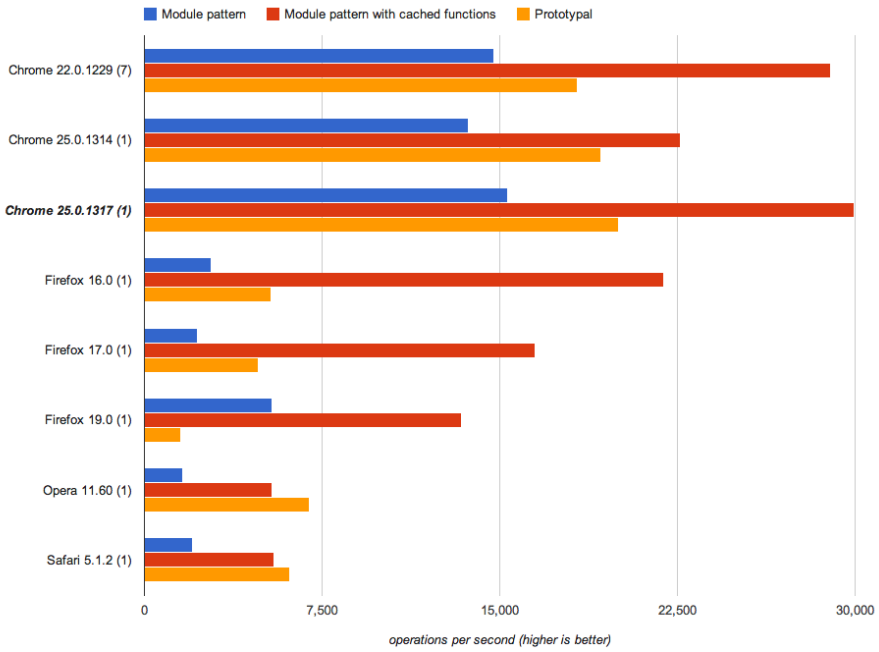
When you absolutely do need to copy objects in a performance-critical code path (and you can't get out of this situation), use an array or a custom "copy constructor" function which copies each property explicitly. This is probably the fastest way to do it:

```
function clone(original) {
  this.foo = original.foo;
  this.bar = original.bar;
}
var copy = new clone(original);
```

Cached Functions in the Module Pattern

Caching your functions when using the module pattern can lead to performance improvements. See below for an example where the variation you're probably used to seeing is slower as it forces new copies of the member functions to be created all the time.

178. http://jsperf.com/object-create-vs-constructor-vs-object-literal/7



Performance improvements when using the module or prototypal patterns.

Here is a test of prototype versus module pattern performance[179]

```
// Prototypal pattern
Klass1 = function () {}
Klass1.prototype.foo = function () {
    log('foo');
}
Klass1.prototype.bar = function () {
    log('bar');
}

// Module pattern
Klass2 = function () {
    var foo = function () {
        log('foo');
    },
    bar = function () {
        log('bar');
    };

    return {
        foo: foo,
```

179. http://jsperf.com/prototypal-performance/12

```
        bar: bar
    }
}


// Module pattern with cached functions
var FooFunction = function () {
    log('foo');
};
var BarFunction = function () {
    log('bar');
};

Klass3 = function () {
    return {
        foo: FooFunction,
        bar: BarFunction
    }
}


// Iteration tests

// Prototypal
var i = 1000,
    objs = [];
while (i--) {
    var o = new Klass1()
    objs.push(new Klass1());
    o.bar;
    o.foo;
}

// Module pattern
var i = 1000,
    objs = [];
while (i--) {
    var o = Klass2()
    objs.push(Klass2());
    o.bar;
    o.foo;
```

```
    }

    // Module pattern with cached functions
    var i = 1000,
        objs = [];
    while (i--) {
        var o = Klass3()
        objs.push(Klass3());
        o.bar;
        o.foo;
    }
// See the test for full details
```

Note: If you don't require a class, avoid the trouble of creating one. Here's[180] an example of how to gain performance boosts by removing the class overhead altogether.

TIPS WHEN USING ARRAYS

Next let's look at a few tips for arrays. In general, **don't delete array elements**. It would make the array transition to a slower internal representation. When the key set becomes sparse, V8 will eventually switch elements to dictionary mode, which is even slower.

Array Literals

Array literals are useful because they give a hint to the VM about the size and type of the array. They're typically good for small to medium sized arrays.

```
// Here V8 can see that you want a 4-element array
containing numbers:
var a = [1, 2, 3, 4];

// Don't do this:
a = []; // Here V8 knows nothing about the array
for(var i = 1; i <= 4; i++) {
     a.push(i);
}
```

180. http://jsperf.com/prototypal-performance/54

Storage of Single Types Vs. Mixed Types

It's never a good idea to mix values of different types (e.g. numbers, strings, undefined or true/false) in the same array (i.e. `var arr = [1, "1", undefined, true, "true"]`)

Test of type inference performance[181]

As we can see from the results, the array of `ints` is the fastest.

Sparse Arrays vs. Full Arrays

When you use sparse arrays, be aware that accessing elements in them is much slower than in full arrays. That's because V8 doesn't allocate a flat backing store for the elements if only a few of them are used. Instead, it manages them in a dictionary, which saves space, but costs time on access.

Test of sparse arrays versus full arrays[182].

The full array `sum` and `sum` of all elements on an array without zeros were actually the fastest. Whether the full array contains zeroes or not should not make a difference.

Packed Vs. Holey Arrays

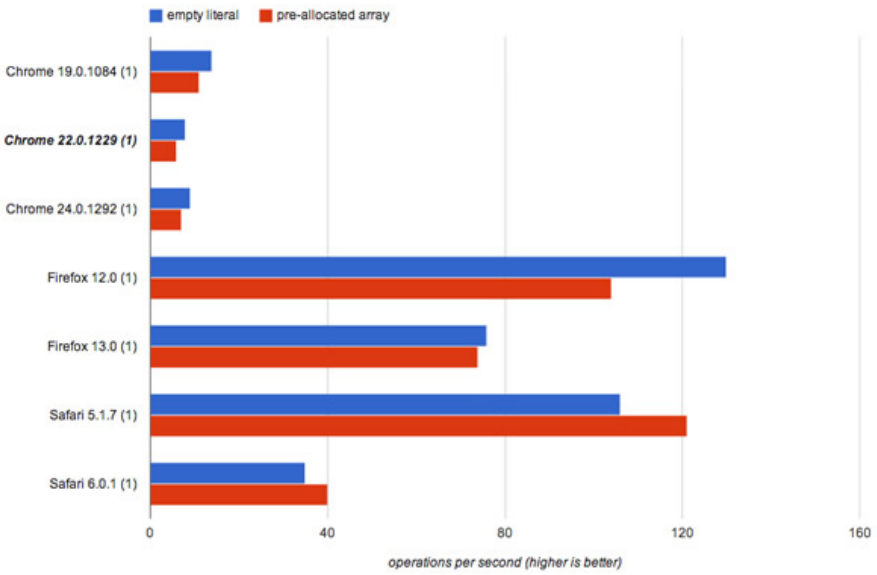
Avoid "holes" in an array (created by deleting elements or `a[x] = foo` with `x > a.length`). Even if only a single element is deleted from an otherwise "full" array, things will be much slower.

Test of packed versus holey arrays[183].

Pre-allocating Arrays Vs. Growing As You Go

Don't pre-allocate large arrays (i.e. greater than 64K elements) to their maximum size, instead grow as you go. Before we get to the performance tests for this tip, keep in mind that this is specific to only some JavaScript engines.

181. http://jsperf.com/type-inference-performance/2
182. http://jsperf.com/sparse-arrays-vs-full-arrays
183. http://jsperf.com/packed-vs-holey-arrays



Test of empty literal versus pre-allocated array in various browsers.

Nitro (Safari) actually treats pre-allocated arrays more favorably. However, in other engines (V8, SpiderMonkey), not pre-allocating is more efficient.

Test of pre-allocated arrays[184].

```
// Empty array
var arr = [];
for (var i = 0; i < 1000000; i++) {
    arr[i] = i;
}

// Pre-allocated array
var arr = new Array(1000000);
for (var i = 0; i < 1000000; i++) {
    arr[i] = i;
}
```

Optimizing Your Application

In the world of Web applications, **speed is everything**. No user wants a spreadsheet application to take seconds to sum up an entire column or a summary of their messages to take a minute before it's ready. This

184. http://jsperf.com/pre-allocated-arrays

is why squeezing every drop of extra performance you can out of code can sometimes be critical.

Image source: Per Olof Forsberg[185].

While understanding and improving your application performance is useful, it can also be difficult. We recommend the following steps to fix performance pain points:

- Measure it: Find the slow spots in your application (~45%)
- Understand it: Find out what the actual problem is (~45%)
- Fix it! (~10%)

Some of the tools and techniques recommended below can assist with this process.

BENCHMARKING

There are many ways to run benchmarks on JavaScript snippets to test their performance—the general assumption being that benchmarking is simply comparing two timestamps. One such pattern was pointed

185. http://www.flickr.com/photos/perolofforsberg/6691744587/in/photostream/

out by the jsPerf[186] team, and happens to be used in SunSpider[187]'s and Kraken[188]'s benchmark suites:

```
var totalTime,
    start = new Date,
    iterations = 1000;
while (iterations--) {
  // Code snippet goes here
}
// totalTime → the number of milliseconds taken
// to execute the code snippet 1000 times
totalTime = new Date - start;
```

Here, the code to be tested is placed within a loop and run a set number of times (e.g. six). After this, the start date is subtracted from the end date to find the time taken to perform the operations in the loop.

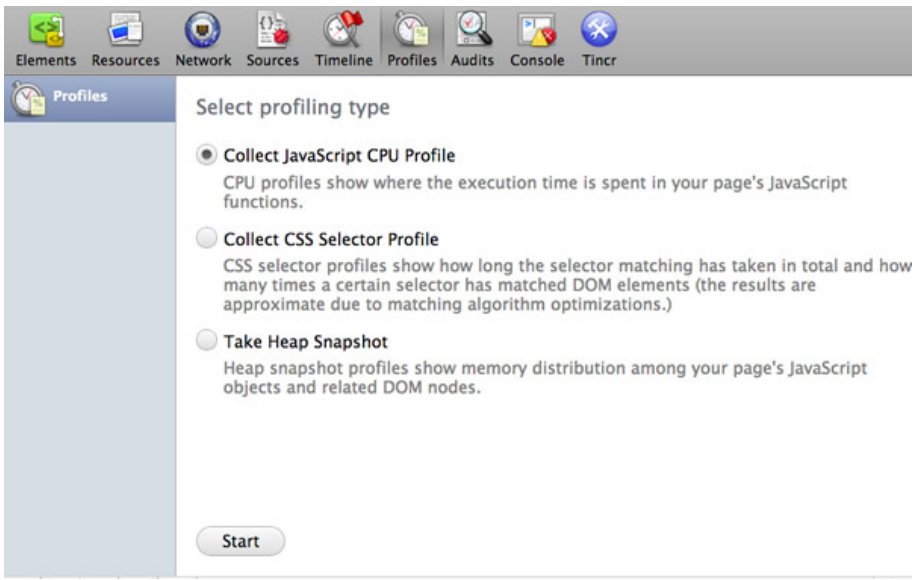
However, this oversimplifies how benchmarking should be done, especially if you want to run the benchmarks in multiple browsers and environments. Garbage collection itself can have an impact on your results. Even if you're using a solution like `window.performance`, you still have to account for these pitfalls.

Regardless of whether you are simply running benchmarks against parts of your code, writing a test suite or coding a benchmarking library, there's a lot more to JavaScript benchmarking than you might think. For a more detailed guide to benchmarking, I highly recommend reading JavaScript Benchmarking[189] by Mathias Bynens and John-David Dalton.

PROFILING

The Chrome Developer Tools have good support for JavaScript profiling[190]. You can use this feature to detect what functions are eating up the most of your time so that you can then go optimize them. This is important, as even small changes to your codebase can have serious impacts on your overall performance.

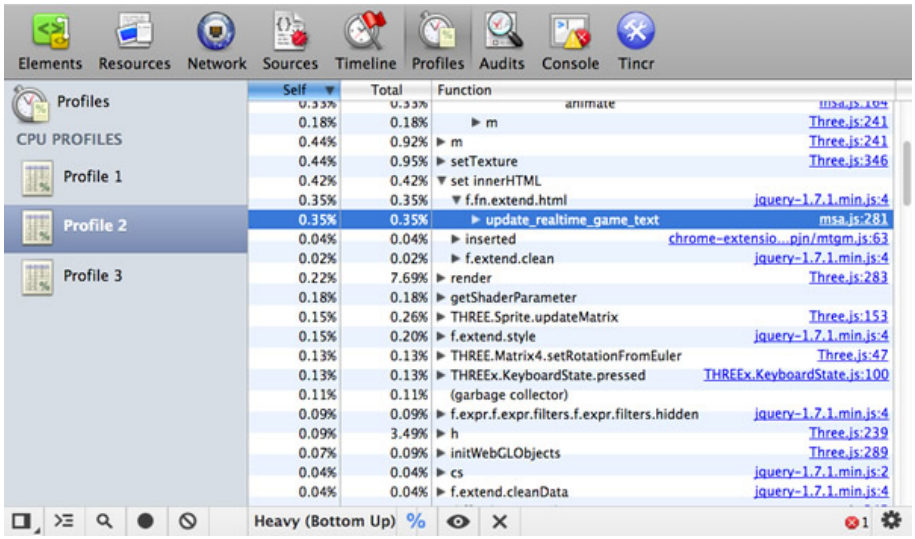
186. http://jsperf.com
187. http://www.webkit.org/perf/sunspider/sunspider.html
188. http://krakenbenchmark.mozilla.org/
189. http://mathiasbynens.be/notes/javascript-benchmarking
190. https://developers.google.com/chrome-developer-tools/docs/profiles



Profiles Panel in Chrome Developer Tools.

Profiling starts with obtaining a baseline for your code's current performance, which can be discovered using the Timeline. This will tell us how long our code took to run. The Profiles tab then gives us a better view into what's happening in our application. The JavaScript CPU profile shows us how much CPU time is being used by our code, the CSS selector profile shows us how much time is spent processing selectors and Heap snapshots show how much memory is being used by our objects.

Using these tools, we can isolate, tweak and reprofile to gauge whether changes we're making to specific functions or operations are improving performance.



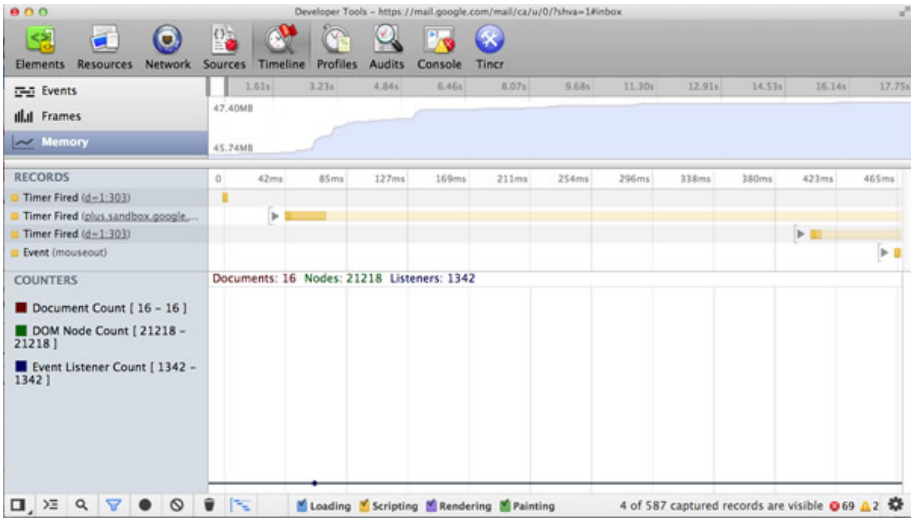
The Profile tab gives you information about your code's performance.

For a good introduction to profiling, read JavaScript Profiling With The Chrome Developer Tools[191], by Zack Grossbart.

Tip: Ideally, you want to ensure that your profiling isn't being affected by extensions or applications you've installed, so run Chrome using the `--user-data-dir <empty_directory>` flag. Most of the time, this approach to optimization testing should be enough, but there are times when you need more. This is where V8 flags can be of help.

AVOIDING MEMORY LEAKS—THREE SNAPSHOT TECHNIQUES FOR DISCOVERY

Internally at Google, the Chrome Developer Tools are heavily used by teams such as Gmail to help us discover and squash memory leaks.



Memory statistics in Chrome Developer Tools.

Some of the memory statistics that our teams care about include private memory usage, JavaScript heap size, DOM node counts, storage clearing, event listener counts and what's going on with garbage collection. For those familiar with event-driven architectures, you might be interested to know that one of the most common issues we used to have were `listen()`'s without `unlisten()`'s (Closure) and missing `dispose()`'s for objects that create event listeners.

Luckily the DevTools can help locate some of these issues, and Loreena Lee has a fantastic presentation available documenting the "3

191. http://coding.smashingmagazine.com/2012/06/12/javascript-profiling-chrome-developer-tools/

snapshot" technique[192] for finding leaks within the DevTools that I can't recommend reading through enough.

The gist of the technique is that you record a number of actions in your application, force a garbage collection, check if the number of DOM nodes doesn't return to your expected baseline and then analyze three heap snapshots to determine if you have a leak.

MEMORY MANAGEMENT IN SINGLE-PAGE APPLICATIONS

Memory management is quite important when writing modern single-page applications (e.g. AngularJS, Backbone, Ember) as they almost never get refreshed. This means that memory leaks can become apparent quite quickly. This is a huge trap on mobile single-page applications, because of limited memory, and on long-running applications like email clients or social networking applications. **With great power comes great responsibility.**

There are various ways to prevent this. In Backbone, ensure you always dispose old views and references using `dispose()` (currently available in Backbone (edge)[193]). This function was recently added, and removes any handlers added in the view's 'events' object, as well as any collection or model listeners where the view is passed as the third argument (callback context). `dispose()` is also called by the view's `remove()`, taking care of the majority of basic memory cleanup needs when the element is cleared from the screen[194]. Other libraries like Ember clean up observers[195] when they detect that elements have been removed from view to avoid memory leaks.

Some sage advice from Derick Bailey:

> *"Other than being aware of how events work in terms of references, just follow the standard rules for manage memory in JavaScript and you'll be fine. If you are loading data in to a Backbone collection full of User objects you want that collection to be cleaned up so it's not using anymore memory, you must remove all references to the collection and the individual objects in it. Once you remove all references, things will be cleaned up. This is just the standard JavaScript garbage collection rule."*

192. https://docs.google.com/presentation/d/1wUVmf78gG-ra5aOxvTfYdiLkdGaR9OhXRnOlIcEmu2s/pub?start=false&loop=false&delayms=3000#slide=id.g1d65bdf6_0_0
193. https://github.com/documentcloud/backbone/blob/master/backbone.js#L1234
194. https://github.com/documentcloud/backbone/blob/master/backbone.js#L1235
195. https://github.com/emberjs/ember.js/blob/d8f76a7fdde741ae3d1e07b12df9cb6718170e48/packages/ember-handlebars/lib/helpers/binding.js#L296

In his article, Derick covers many of the common memory pitfalls[196] when working with Backbone.js and how to fix them.

There is also a helpful tutorial available for debugging memory leaks in Node[197] by Felix Geisendörfer worth reading, especially if it forms a part of your broader SPA stack.

MINIMIZING REFLOWS

When a browser has to recalculate the positions and geometrics of elements in a document for the purpose of re-rendering it, we call this reflow[198]. Reflow is a user-blocking operation in the browser, so it's helpful to understand how to improve reflow time.

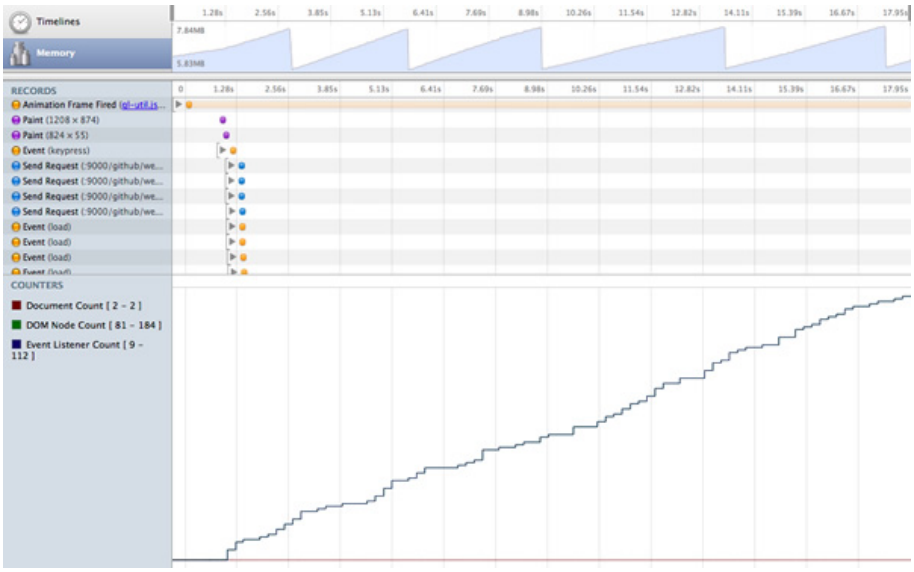


Chart of reflow time.

You should batch methods that trigger reflow[199] or that repaint, and use them sparingly. It's important to process off DOM where possible. This is possible using DocumentFragment[200], a lightweight document object. Think of it as a way to extract a portion of a document's tree, or create a new "fragment" of a document. Rather than constantly adding to the DOM using nodes, we can use document fragments to build up all we need and only perform a single insert into the DOM to avoid excessive reflow.

196. http://lostechies.com/derickbailey/2012/03/19/backbone-js-and-javascript-garbage-collection/
197. https://github.com/felixge/node-memory-leak-tutorial
198. https://www.youtube.com/watch?feature=player_embedded&v=ZHxbs5WEQzE
199. http://stackoverflow.com/questions/510213/when-does-reflow-happen-in-a-dom-environment
200. http://www.w3.org/TR/DOM-Level-2-Core/core.html#ID-B63ED1A3

For example, let's write a function that adds 20 `div`s to an element. Simply appending each new `div` directly to the element could trigger 20 reflows.

```
function addDivs(element) {
  var div;
  for (var i = 0; i < 20; i ++) {
    div = document.createElement('div');
    div.innerHTML = 'Heya!';
    element.appendChild(div);
  }
}
```

To work around this issue, we can use `DocumentFragment`, and instead, append each of our new `div`s to this. When appending to the `DocumentFragment` with a method like `appendChild`, all of the fragment's children are appended to the element triggering only one reflow.

```
function addDivs(element) {
  var div;
  // Creates a new empty DocumentFragment.
  var fragment = document.createDocumentFragment();
  for (var i = 0; i < 20; i ++) {
    div = document.createElement('a');
    div.innerHTML = 'Heya!';
    fragment.appendChild(div);
  }
  element.appendChild(fragment);
}
```

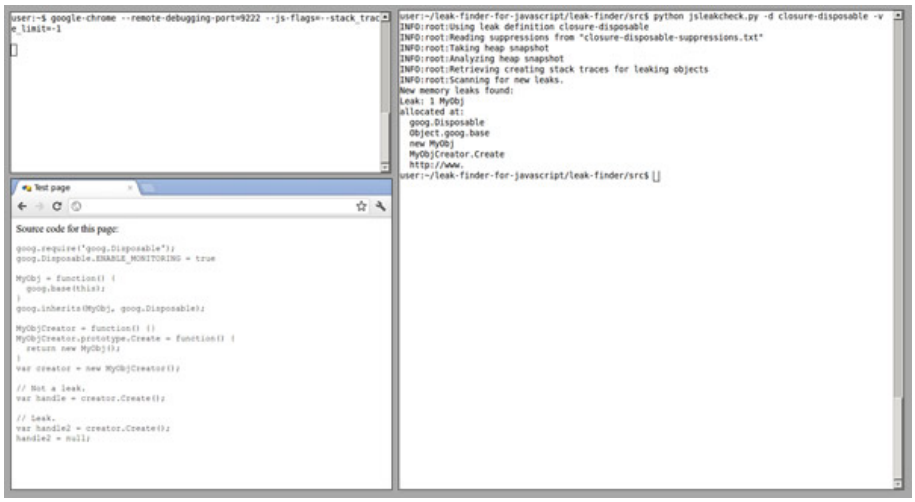
You can read more about this topic at Make the Web Faster[201], JavaScript Memory Optimization[202] and Finding Memory Leaks[203].

JAVASCRIPT MEMORY LEAK DETECTOR

To help discover JavaScript memory leaks, two of my fellow Googlers (Marja Hölttä and Jochen Eisinger) developed a tool that works with the Chrome Developer Tools (specifically, the remote inspection protocol),

201. https://developers.google.com/speed/articles/javascript-dom
202. http://blog.tojicode.com/2012/03/javascript-memory-optimization-and.html
203. http://gent.ilcore.com/2011/08/finding-memory-leaks.html

and retrieves heap snapshots and detects what objects are causing leaks.



A tool for detecting JavaScript memory leaks.

There's a whole post on how to use the tool[204], and I encourage you to check it out or view the Leak Finder project page[205].

Some more information: In case you're wondering why a tool like this isn't already integrated with our Developer Tools, the reason is twofold. It was originally developed to help us catch some specific memory scenarios in the Closure Library, and it makes more sense as an external tool (or maybe even an extension if we get a heap profiling extension API in place).

V8 FLAGS FOR DEBUGGING OPTIMIZATIONS & GARBAGE COLLECTION

Chrome supports passing a number of flags directly to V8 via the `js-flags` flag to get more detailed output about what the engine is optimizing. For example, this traces V8 optimizations:

```
"/Applications/Google Chrome/Google Chrome"
--js-flags="--trace-opt --trace-deopt"
```

Windows users will want to run `chrome.exe --js-flags="--trace-opt --trace-deopt"`

When developing your application, the following V8 flags can be used.

---

204. http://google-opensource.blogspot.de/2012/08/leak-finder-new-tool-for-javascript.html
205. http://code.google.com/p/leak-finder-for-javascript/

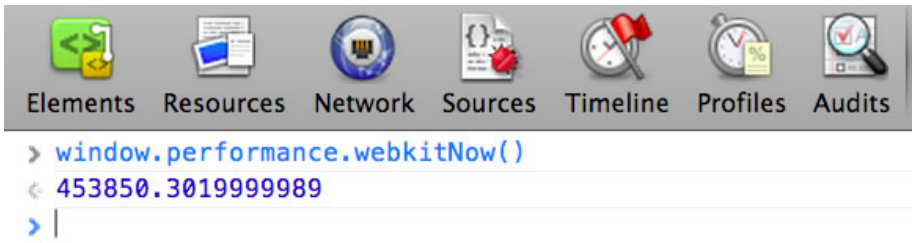
- `trace-opt` – log names of optimized functions and show where the optimizer is skipping code because it can't figure something out.
- `trace-deopt` – log a list of code it had to deoptimize while running.
- `trace-gc` – logs a tracing line on each garbage collection.

V8's tick-processing scripts mark optimized functions with an `*` (asterisk) and non-optimized functions with `~` (tilde).

If you're interested in learning more about V8's flags and how V8's internals work in general, I strongly recommend looking through Vyacheslav Egorov's excellent post on V8 internals[206], which summarizes the best resources available on this at the moment.

HIGH-RESOLUTION TIME AND NAVIGATION TIMING API

High Resolution Time[207] (HRT) is a JavaScript interface providing the current time in sub-millisecond resolution that isn't subject to system clock skews or user adjustments. Think of it as a way to measure more precisely than we've previously had with `new Date` and `Date.now()`. This is helpful when we're writing performance benchmarks.



High Resolution Time (HRT) provides the current time in sub-millisecond resolution.

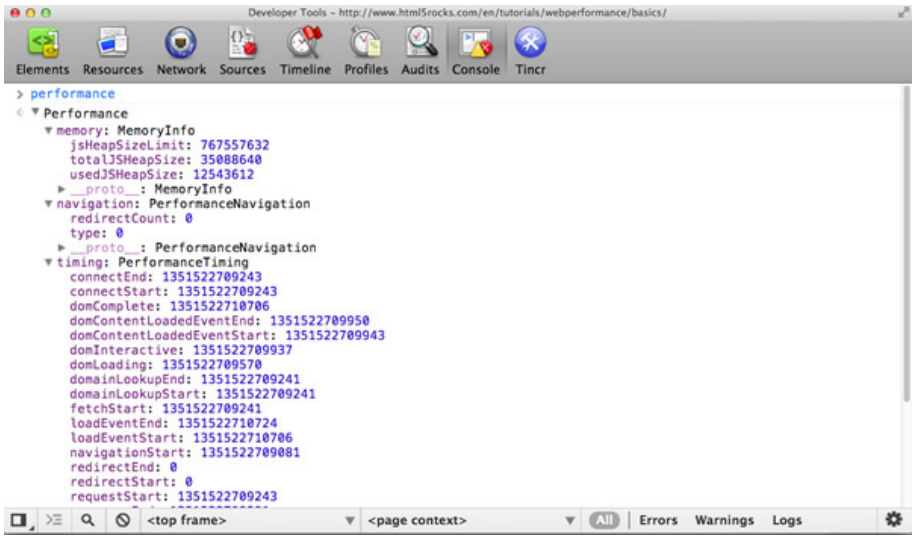
HRT is currently available in Chrome (stable) as `window.performance.webkitNow()`, but the prefix is dropped in Chrome Canary, making it available via `window.performance.now()`. Paul Irish has written more about HRT[208] in a post on HTML5Rocks.

So, we now know the current time, but what if we wanted an API for accurately measuring performance on the Web?

Well, one is now also available in the Navigation Timing API[209]. This API provides a simple way to get accurate and detailed time mea-

206. http://mrale.ph/blog/2011/12/18/v8-optimization-checklist.html
207. http://www.w3.org/TR/hr-time/
208. http://updates.html5rocks.com/2012/08/When-milliseconds-are-not-enough-performance-now
209. http://dvcs.w3.org/hg/webperf/raw-file/tip/specs/NavigationTiming/Overview.html

surements that are recorded while a webpage is loaded and presented to the user. Timing information is exposed via `window.performance.timing`, which you can simply use in the console:



Timing information is shown in the console.

Looking at the data above, we can extract some very useful information. For example, network latency is `responseEnd-fetchStart`, the time taken for a page load once it's been received from the server is `loadEventEnd-responseEnd` and the time taken to process between navigation and page load is `loadEventEnd-navigationStart`.

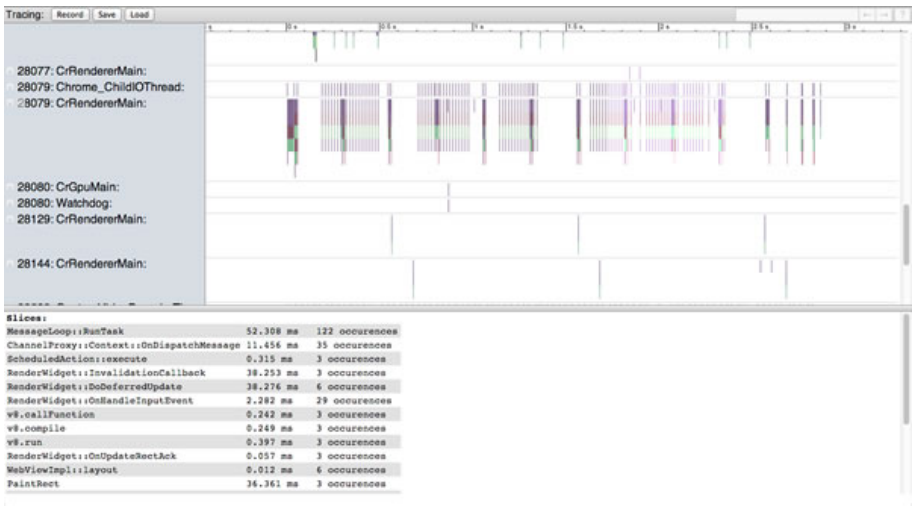
As you can see above, a `perfomance.memory` property is also available that gives access to JavaScript memory usage data such as the total heap size.

For more details on the Navigation Timing API, read Sam Dutton's great article Measuring Page Load Speed With Navigation Timing[210].

ABOUT:MEMORY AND ABOUT:TRACING

`about:tracing` in Chrome offers an intimate view of the browser's performance, recording all of Chrome's activities across every thread, tab and process.

210. http://www.html5rocks.com/en/tutorials/webperformance/basics/



`About:Tracing` *offers an intimate view of the browser's performance.*

What's really useful about this tool is that it allows you to capture profiling data about what Chrome is doing under the hood, so you can properly adjust your JavaScript execution, or optimize your asset loading.

Lilli Thompson has an excellent write-up for games developers[211] on using `about:tracing` to profile WebGL games. The write-up is also useful for general JavaScripters.

Navigating to `about:memory` in Chrome is also useful as it shows the exact amount of memory being used by each tab, which is helpful for tracking down potential leaks.

Conclusion

As we've seen, there are many hidden performance gotchas in the world of JavaScript engines, and no silver bullet available to improve performance. It's only when you combine a number of optimizations in a (real-world) testing environment that you can realize the largest performance gains. But even then, understanding how engines interpret and optimize your code can give you insights to help tweak your applications.

Measure It. Understand it. Fix it. Rinse and repeat.

211. http://www.html5rocks.com/en/tutorials/games/abouttracing/

Image source: Sally Hunter[212].

Remember to care about optimization, but stop short of opting for micro-optimization at the cost of convenience. For example, some developers opt for `.forEach` and `Object.keys` over `for` and `for in` loops, even though they're slower, for the convenience of being able to scope. Do make sanity calls on what optimizations your application absolutely needs and which ones it could live without.

Also, be aware that although JavaScript engines continue to get faster, the next real bottleneck is the DOM. Reflows and repaints are just as important to minimize, so remember to only touch the DOM if it's absolutely required. And do care about networking. HTTP requests are precious, especially on mobile, and you should be using HTTP caching to reduce the size of assets.

Keeping all of these in mind will ensure that you get the most out of the information from this post. I hope you found it helpful!

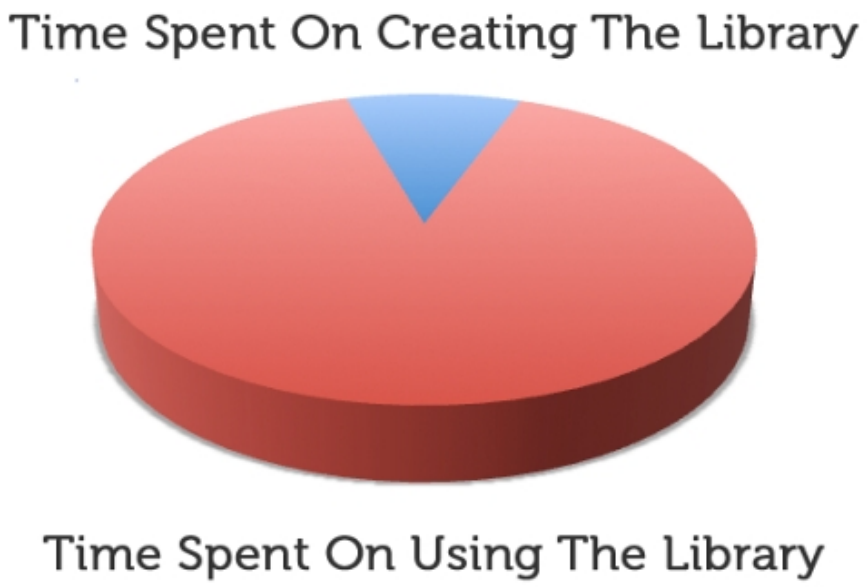
This chapter was reviewed by Jakob Kummerow, Michael Starzinger, Sindre Sorhus, Mathias Bynens, John-David Dalton and Paul Irish. ❧

212. http://www.flickr.com/photos/38891164@N02/4266609887/

Designing Better JavaScript APIs

BY RODNEY REHM

At some point or another, you will find yourself writing JavaScript code that exceeds the couple of lines from a jQuery plugin. Your code will do a whole lot of things; it will (ideally) be used by many people who will approach your code differently. They have different needs, knowledge and expectations.



This chapter covers the most important things that you will need to consider before and while writing your own utilities and libraries. We'll focus on how to make your code *accessible* to other developers. A couple of topics will be touching upon jQuery for demonstration, yet this chapter is neither about jQuery nor about writing plugins for it.

Peter Drucker once said: "The computer is a moron." Don't write code for morons, write for humans! Let's dive into **designing the APIs that developers will love using**.

Fluent Interface

The Fluent Interface[213] is often referred to as *Method Chaining* (although that's only half the truth). To beginners it looks like the *jQuery style*. While I believe the API style was a key ingredient in jQuery's success, it wasn't invented by them—credits seem to go to Martin Fowler who coined the term[214] back in 2005, roughly a year before jQuery was released. Fowler only gave the thing a name, though—Fluent Interfaces have been around for a much longer time.

Aside from major simplifications, jQuery offered to even out severe browser differences. It has always been the *Fluent Interface* that I have loved most about this extremely successful library. I have come to enjoy this particular API style so much that it became immediately apparent that I wanted this style for URI.js[215], as well. While tuning up the URI.js API, I constantly looked through the jQuery source to find the little tricks that would make my implementation as simple as possible. I found out that I was not alone in this endeavor. Lea Verou[216] created chainvas[217]—a tool to wrap regular getter/setter APIs into sweet fluent interfaces. Underscore's `_.chain()`[218] does something similar. In fact, most of the newer generation libraries support method chaining.

METHOD CHAINING

The general idea of Method Chaining[219] is to achieve code that is as fluently readable as possible and thus quicker to understand. With *Method Chaining* we can form code into sentence-like sequences, making code easy to read, while reducing noise in the process:

```
// regular API calls to change some colors and add an
event-listener
var elem = document.getElementById("foobar");
elem.style.background = "red";
elem.style.color = "green";
elem.addEventListener('click', function(event) {
  alert("hello world!");
}, true);
```

213. http://en.wikipedia.org/wiki/Fluent_interface#JavaScript
214. http://martinfowler.com/bliki/FluentInterface.html
215. http://medialize.github.com/URI.js/
216. https://twitter.com/leaverou
217. http://lea.verou.me/chainvas/
218. http://underscorejs.org/#chain
219. http://en.wikipedia.org/wiki/Method_chaining

```
// (imaginary) method chaining API
DOMHelper.getElementById('foobar')
  .setStyle("background", "red")
  .setStyle("color", "green")
  .addEvent("click", function(event) {
    alert("hello world");
  });
```

Note how we didn't have to assign the element's reference to a variable and repeat that over and over again.

COMMAND QUERY SEPARATION

Command and Query Separation[220] (CQS) is a concept inherited from imperative programming. Functions that change the state (internal values) of an object are called *commands*, functions that retrieve values are called *queries*. In principle, queries return data, commands change the state—but neither does both. This concept is one of the foundations of the everyday getter and setter methods we see in most libraries today. Since *Fluent Interfaces* return a self-reference for chaining method calls, we're already breaking the rule for *commands*, as they are not supposed to return anything. On top of this (easy to ignore) trait, we (deliberately) break with this concept to keep APIs as simple as possible. An excellent example for this practice is jQuery's `css()`[221] method:

```
var $elem = jQuery("#foobar");

// CQS - command
$elem.setCss("background", "green");
// CQS - query
$elem.getCss("color") === "red";

// non-CQS - command
$elem.css("background", "green");
// non-CQS - query
$elem.css("color") === "red";
```

As you can see, getter and setter methods are merged into a single method. The action to perform (namely, *query* or *command*) is decided

220. http://en.wikipedia.org/wiki/Command-query_separation
221. http://api.jquery.com/css/

by the amount of arguments passed to the function, rather than by which function was called. This allows us to expose fewer methods and in turn type less to achieve the same goal.

It is not necessary to compress getters and setters into a single method in order to create a fluid interface—it boils down to personal preference. Your documentation should be very clear with the approach you've decided on. I will get into documenting APIs later, but at this point I would like to note that multiple function signatures may be harder to document.

GOING FLUENT

While method chaining already does most of the job for going fluent, you're not off the hook yet. To illustrate the next step of *fluent*, we're pretending to write a little library handling date intervals. An interval starts with a date and ends with a date. A date is not necessarily connected to an interval. So we come up with this simple constructor:

```
// create new date interval
var interval = new DateInterval(startDate, endDate);
// get the calculated number of days the interval spans
var days = interval.days();
```

While this looks right at first glance, this example shows what's wrong:

```
var startDate = new Date(2012, 0, 1);
var endDate = new Date(2012, 11, 31)
var interval = new DateInterval(startDate, endDate);
var days = interval.days(); // 365
```

We're writing out a whole bunch of variables and stuff we probably won't need. A nice solution to the problem would be to add a function to the Date object in order to return an interval:

```
// DateInterval creator for fluent invocation
Date.prototype.until = function(end) {

  // if we weren't given a date, make one
  if (!(end instanceof Date)) {
    // create date from given arguments,
    // proxy the constructor to allow for any parameters
    // the Date constructor would've taken natively
    end = Date.apply(null,
```

```
        Array.prototype.slice.call(arguments, 0)
      );
    }

    return new DateInterval(this, end);
  };
```

Now we can create that `DateInterval` in a fluent, easy to type-and-read fashion:

```
var startDate = new Date(2012, 0, 1);
var interval = startDate.until(2012, 11, 31);
var days = interval.days(); // 365

// condensed fluent interface call:
var days = (new Date(2012, 0, 1))
  .until(2012, 11, 31) // returns DateInterval instance
  .days(); // 365
```

As you can see in this last example, there are less variables to declare, less code to write, and the operation almost reads like an english sentence. With this example you should have realized that method chaining is only a part of a fluent interface, and as such, the terms are not synonymous. To provide fluency, you have to think about code streams—where are you coming from and where you are headed.

This example illustrated fluidity by extending a native object with a custom function. This is as much a religion as using semicolons or not. In Extending built-in native objects. Evil or not?[222] kangax[223] explains the ups and downs of this approach. While everyone has their opinions about this, the one thing everybody agrees on is keeping things consistent. As an aside, even the followers of "Don't pollute native objects with custom functions" would probably let the following, still somewhat fluid trick slide:

```
String.prototype.foo = function() {
  return new Foo(this);
}
```

222. http://perfectionkills.com/extending-built-in-native-objects-evil-or-not/
223. https://twitter.com/kangax

```
"I'm a native object".foo()
  .iAmACustomFunction();
```

With this approach your functions are still within your namespace, but made accessible through another object. Make sure your equivalent of `.foo()` is a non-generic term, something highly unlikely to collide with other APIs. Make sure you provide proper `.valueOf()`[224] and `.toString()`[225] methods to convert back to the original primitive types.

Consistency

Jake Archibald[226] once had a slide defining *Consistency*. It simply read *Not PHP*[227]. Do. Not. Ever. Find yourself naming functions like *str_repeat()*, *strpos()*, *substr()*. Also, don't ever shuffle around positions of arguments. If you declared `find_in_array(haystack, needle)` at some point, introducing `findInString(needle, haystack)` will invite an angry mob of zombies to rise from their graves to hunt you down and force you to write delphi for the rest of your life!

NAMING THINGS

> *"There are only two hard problems in computer science: cache-invalidation and naming things."*
>
> — *Phil Karlton*

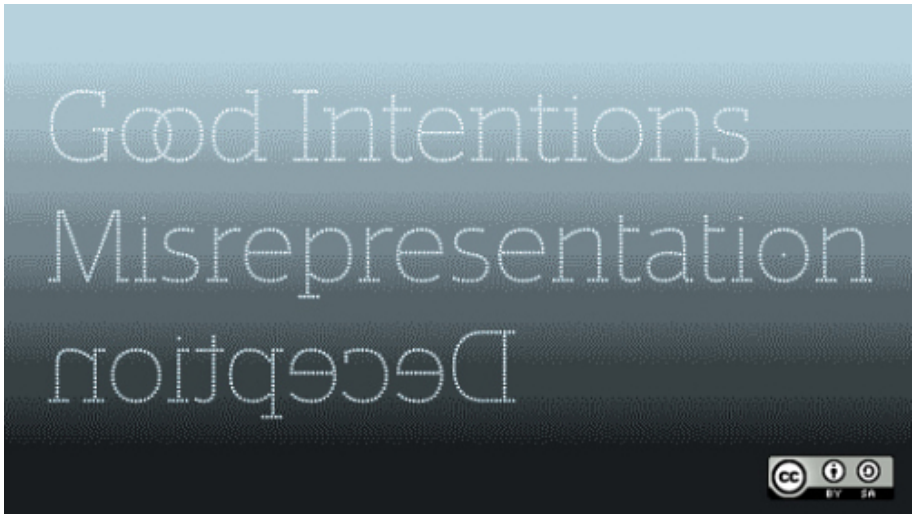
I've been to numerous talks and sessions trying to teach me the finer points of naming things. I haven't left any of them without having heard the above said quote, nor having learnt how to actually name things. My advice boils down to *keep it short but descriptive and go with your gut*. But most of all, keep it consistent.

The `DateInterval` example above introduced a method called `until()`. We could have named that function `interval()`. The latter would have been closer to the returned value, while the former is more *humanly readable*. Find a line of wording you like and stick with it. Consistency is 90% of what matters. Choose one style and keep that

224. https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Object/valueOf
225. https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Global_Objects/Object/toString
226. https://twitter.com/jaffathecake
227. http://www.slideshare.net/slideshow/embed_code/5426258?startSlide=59

style—even if you find yourself disliking it at some point in the future.

Handling Arguments



How your methods accept data is more important than making them chainable. While method chaining is a pretty generic thing that you can easily make your code do, handling arguments is not. You'll need to think about how the methods you provide are most likely going to be used. Is code that uses your API likely to repeat certain function calls? Why are these calls repeated? How could your API help the developer to reduce the noise of repeating method calls?

jQuery's `css()`[228] method can set styles on a DOM element:

```
jQuery("#some-selector")
  .css("background", "red")
  .css("color", "white")
  .css("font-weight", "bold")
  .css("padding", 10);
```

There's a pattern here! Every method invocation is naming a style and specifying a value for it. This calls for having the method accept a map:

```
jQuery("#some-selector").css({
  "background" : "red",
  "color" : "white",
  "font-weight" : "bold",
```

228. http://api.jquery.com/css/

```
  "padding" : 10
});
```

jQuery's `on()`[229] method can register event handlers. Like `css()` it accepts a map of events, but takes things even further by allowing a single handler to be registered for multiple events:

```
// binding events by passing a map
jQuery("#some-selector").on({
  "click" : myClickHandler,
  "keyup" : myKeyupHandler,
  "change" : myChangeHandler
});

// binding a handler to multiple events:
jQuery("#some-selector").on("click keyup change",
myEventHandler);
```

You can offer the above function signatures by using the following *method pattern*:

```
DateInterval.prototype.values = function(name, value) {
  var map;

  if (jQuery.isPlainObject(name)) {
    // setting a map
    map = name;
  } else if (value !== undefined) {
    // setting a value (on possibly multiple names),
convert to map
    keys = name.split(" ");
    map = {};
    for (var i = 0, length = keys.length; i < length;
i++) {
      map[keys[i]] = value;
    }
  } else if (name === undefined) {
    // getting all values
    return this.values;
```

229. http://api.jquery.com/on/

```
  } else {
    // getting specific value
    return this.values[name];
  }

  for (var key in map) {
    this.values[name] = map[key];
  }

  return this;
};
```

If you are working with collections, think about what you can do to reduce the number of loops an API user would probably have to make. Say we had a number of `<input>` elements for which we want to set the default value:

```
<input type="text" value="" data-default="foo">
<input type="text" value="" data-default="bar">
<input type="text" value="" data-default="baz">
```

We'd probably go about this with a loop:

```
jQuery("input").each(function() {
  var $this = jQuery(this);
  $this.val($this.data("default"));
});
```

What if we could bypass that method with a simple callback that gets applied to each `<input>` in the collection? jQuery developers have thought of that and allow us to write less™:

```
jQuery("input").val(function() {
  return jQuery(this).data("default");
});
```

It's the little things like accepting maps, callbacks or serialized attribute names, that make using your API not only cleaner, but more comfortable and efficient to use. Obviously not all of your APIs' methods will benefit from this method pattern—it's up to you to decide where all this makes sense and where it is just a waste of time. Try to be as consistent about this as humanly possible. *Reduce the need for boilerplate code with the tricks shown above and people will invite you over for a drink.*

HANDLING TYPES

Whenever you define a function that will accept arguments, you decide what data that function accepts. A function to calculate the number of days between two dates could look like:

```
DateInterval.prototype.days = function(start, end) {
  return Math.floor((end - start) / 86400000);
};
```

As you can see, the function expects numeric input—a millisecond timestamp, to be exact. While the function does what we intended it to do, it is not very versatile. What if we're working with `Date` objects or a string representation of a date? Is the user of this function supposed to cast data all the time? No! Simply verifying the input and casting it to whatever we need it to be should be done in a central place, not cluttered throughout the code using our API:

```
DateInterval.prototype.days = function(start, end) {
  if (!(start instanceof Date)) {
    start = new Date(start);
  }
  if (!(end instanceof Date)) {
    end = new Date(end);
  }

  return Math.floor((end.getTime() - start.getTime()) /
86400000);
};
```

By adding these six lines we've given the function the power to accept a Date object, a numeric timestamp, or even a string representation like `Sat Sep 08 2012 15:34:35 GMT+0200 (CEST)`. We do not know how and for what people are going to use our code, but with a little foresight, we can make sure there is little pain with integrating our code.

The experienced developer can spot another problem in the example code. We're assuming `start` comes before `end`. If the API user accidentally swapped the dates, he'd be given a negative value for the number of days between `start` and `end`. Stop and think about these situations carefully. If you've come to the conclusion that a negative value doesn't make sense, fix it:

```
DateInterval.prototype.days = function(start, end) {
  if (!(start instanceof Date)) {
```

```
    start = new Date(start);
  }
  if (!(end instanceof Date)) {
    end = new Date(end);
  }

  return Math.abs(Math.floor((end.getTime() -
start.getTime()) / 86400000));
};
```

JavaScript allows type casting a number of ways. If you're dealing with primitives (string, number, boolean) it can get as simple (as in "short") as:

```
function castaway(some_string, some_integer,
some_boolean) {
  some_string += "";
  some_integer += 0; // parseInt(some_integer, 10) is
the safer bet
  some_boolean = !!some_boolean;
}
```

I'm not advocating you to do this everywhere and at all times. But these innocent looking lines may save time and some suffering while integrating your code.

TREATING UNDEFINED AS AN EXPECTED VALUE

There will come a time when `undefined` is a value that your API actually expects to be given for setting an attribute. This might happen to "unset" an attribute, or simply to gracefully handle bad input, making your API more robust. To identify if the value `undefined` has actually been passed by your method, you can check the `arguments`[230] object:

```
function testUndefined(expecting, someArgument) {
  if (someArgument === undefined) {
    console.log("someArgument was undefined");
  }
  if (arguments.length > 1) {
```

230. https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Functions_and_function_scope/arguments

```
    console.log("but was actually passed in");
  }
}

testUndefined("foo");
// prints: someArgument was undefined
testUndefined("foo", undefined);
// prints: someArgument was undefined, but was actually
passed in
```

NAMED ARGUMENTS

```
event.initMouseEvent(
  "click", true, true, window,
  123, 101, 202, 101, 202,
  true, false, false, false,
  1, null);
```

The function signature of Event.initMouseEvent[231] is a nightmare come true. There is no chance any developer will remember what that 1 (second to last parameter) means without looking it up in the documentation. No matter how good your documentation is, do what you can so people don't have to look things up!

HOW OTHERS DO IT

Looking beyond our beloved language, we find Python knowing a concept called named arguments[232]. It allows you to declare a function providing default values for arguments, allowing your attributed names to be stated in the calling context:

```
function namesAreAwesome(foo=1, bar=2) {
  console.log(foo, bar);
}

namesAreAwesome();
// prints: 1, 2
```

231. https://developer.mozilla.org/en-US/docs/DOM/event.initMouseEvent
232. http://www.diveintopython.net/power_of_introspection/optional_arguments.html

```
namesAreAwesome(3, 4);
// prints: 3, 4

namesAreAwesome(foo=5, bar=6);
// prints: 5, 6

namesAreAwesome(bar=6);
// prints: 1, 6
```

Given this scheme, initMouseEvent() could've looked like a self-explaining function call:

```
event.initMouseEvent(
  type="click",
  canBubble=true,
  cancelable=true,
  view=window,
  detail=123,
  screenX=101,
  screenY=202,
  clientX=101,
  clientY=202,
  ctrlKey=true,
  altKey=false,
  shiftKey=false,
  metaKey=false,
  button=1,
  relatedTarget=null);
```

In JavaScript this is not possible today. While "the next version of JavaScript" (frequently called ES.next, ES6, or Harmony) will have default parameter values[233] and rest parameters[234], there is still no sign of named parameters.

ARGUMENT MAPS

JavaScript not being Python (and ES.next being light years away), we're left with fewer choices to overcome the obstacle of "argument forests". jQuery (and pretty much every other decent API out there) chose to

233. http://wiki.ecmascript.org/doku.php?id=harmony:parameter_default_values
234. http://wiki.ecmascript.org/doku.php?id=harmony:rest_parameters

work with the concept of "option objects". The signature of jQuery.ajax()[235] provides a pretty good example. Instead of accepting numerous arguments, we only accept an object:

```
function nightmare(accepts, async, beforeSend, cache,
complete, /* and 28 more */) {
  if (accepts === "text") {
    // prepare for receiving plain text
  }
}

function dream(options) {
  options = options || {};
  if (options.accepts === "text") {
    // prepare for receiving plain text
  }
}
```

Not only does this prevent insanely long function signatures, it also makes calling the function more descriptive:

```
nightmare("text", true, undefined, false, undefined, /*
and 28 more */);

dream({
  accepts: "text",
  async: true,
  cache: false
});
```

Also, we do not have to touch the function signature (adding a new argument) should we introduce a new feature in a later version.

DEFAULT ARGUMENT VALUES

jQuery.extend()[236], _.extend()[237] and Protoype's Object.extend[238] are functions that let you merge objects, allowing you to throw your own preset options object into the mix:

---

235. http://api.jquery.com/jquery.ajax/
236. http://api.jquery.com/jQuery.extend/
237. http://underscorejs.org/#extend
238. http://api.prototypejs.org/language/Object/extend/

```
var default_options = {
  accepts: "text",
  async: true,
  beforeSend: null,
  cache: false,
  complete: null,
  // …
};

function dream(options) {
  var o = jQuery.extend({}, default_options, options ||
{});
  console.log(o.accepts);
}

// make defaults public
dream.default_options = default_options;

dream({ async: false });
// prints: "text"
```

You're earning bonus points for making the default values publicly accessible. With this, anyone can change `accepts` to "json" in a central place, and thus avoid specifying that option over and over again. Note that the example will always append `|| {}` to the initial read of the option object. This allows you to call the function without an argument given.

GOOD INTENTIONS—A.K.A. "PITFALLS"

Now that you know how to be truly flexible in accepting arguments, we need to come back to an old saying:

> *"With great power comes great responsibility!"*
> *— Voltaire*

As with most weakly-typed languages, JavaScript does automatic casting when it needs to. A simple example is testing the truthfulness:

```
var foo = 1;
var bar = true;

if (foo) {
```

```
  // yep, this will execute
}

if (bar) {
  // yep, this will execute
}
```

We're quite used to this automatic casting. We're so used to it, that we forget that although something is truthful, it may not be the boolean truth. Some APIs are so flexible they are *too smart* for their own good. Take a look at the signatures of jQuery.toggle()[239]:

```
.toggle( /* int */ [duration] [, /* function */
callback] )
.toggle( /* int */ [duration] [, /* string */  easing]
[, /* function */ callback] )
.toggle( /* bool */ showOrHide )
```

It will take us some time decrypting why these behave *entirely* different:

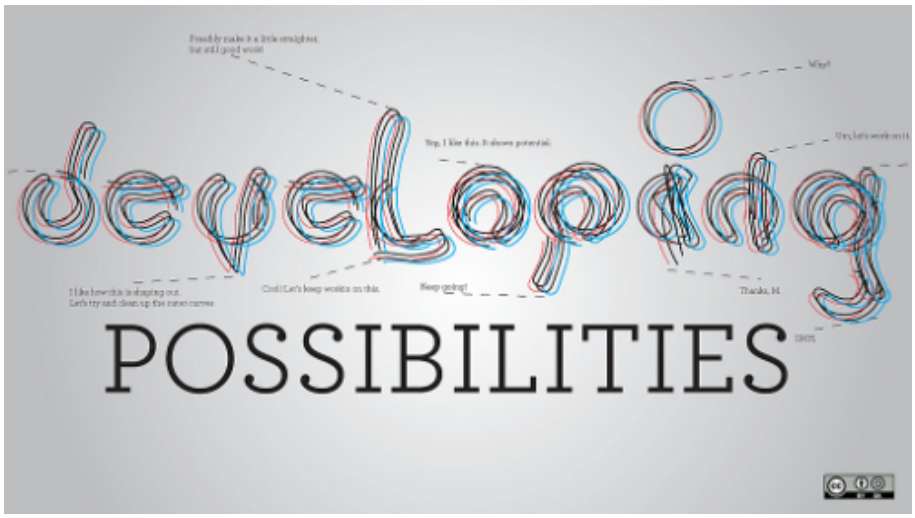
```
var foo = 1;
var bar = true;
var $hello = jQuery(".hello");
var $world = jQuery(".world");

$hello.toggle(foo);
$world.toggle(bar);
```

We were *expecting* to use the `showOrHide` signature in both cases. But what really happened is `$hello` doing a toggle with a `duration` of one millisecond. This is not a bug in jQuery, this is a simple case of *expectation not met*. Even if you're an experienced jQuery developer, you *will* trip over this from time to time.

You are free to add as much convenience / sugar as you like—but do not sacrifice a clean and (mostly) robust API along the way. If you find yourself providing something like this, think about providing a separate method like `.toggleIf(bool)` instead. Whatever choice you make, keep your API consistent!

239. http://api.jquery.com/toggle/

Extensibility



With option objects, we've covered the topic of extensible configuration. Let's talk about allowing the API user to extend the core and API itself. This is an important topic, as it allows your code to focus on the important things, while having API users implement edge-cases themselves. Good APIs are concise APIs. Having a hand full of configuration options is fine, but having a couple dozen of them makes your API feel bloated and opaque. Focus on the primary-use cases, only do the things most of your API users will need. Everything else should be left up to them. To allow API users to extend your code to suit their needs, you have a couple of options...

CALLBACKS

Callbacks can be used to achieve extensibility by configuration. You can use callbacks to allow the API user to override certain parts of your code. When you feel specific tasks may be handled differently than your default code, refactor that code into a configurable callback function to allow an API user to easily override that:

```
var default_options = {
  // ...
  position: function($elem, $parent) {
    $elem.css($parent.position());
  }
};

function Widget(options) {
  this.options = jQuery.extend({}, default_options,
```

```
options || {});
  this.create();
};

Widget.prototype.create = function() {
  this.$container =
$("<div></div>").appendTo(document.body);
  this.$thingie =
$("<div></div>").appendTo(this.$container);
  return this;
};

Widget.protoype.show = function() {
  this.options.position(this.$thingie, this.$container);
  this.$thingie.show();
  return this;
};


var widget = new Widget({
  position: function($elem, $parent) {
    var position = $parent.position();
    // position $elem at the lower right corner of
$parent
    position.left += $parent.width();
    position.top += $parent.height();
    $elem.css(position);
  }
});
widget.show();
```

Callbacks are also a generic way to allow API users to customize elements your code has created:

```
// default create callback doesn't do anything
default_options.create = function($thingie){};

Widget.prototype.create = function() {
  this.$container =
$("<div></div>").appendTo(document.body);
  this.$thingie =
$("<div></div>").appendTo(this.$container);
```

```
  // execute create callback to allow decoration
  this.options.create(this.$thingie);
  return this;
};

var widget = new Widget({
  create: function($elem) {
    $elem.addClass('my-style-stuff');
  }
});
widget.show();
```

Whenever you accept callbacks, be sure to document their signature and provide examples to help API users customize your code. Make sure you're consistent about the context (where `this` points to) in which callbacks are executed in, and the arguments they accept.

EVENTS

Events come naturally when working with the DOM. In larger application we use events in various forms (e.g. PubSub) to enable communication between modules. Events are particularly useful and feel most natural when dealing with UI widgets. Libraries like jQuery offer simple interfaces allowing you to easily conquer this domain.

Events interface best when there is something happening—hence the name. Showing and hiding a widget could depend on circumstances outside of your scope. Updating the widget when it's shown is also a very common thing to do. Both can be achieved quite easily using jQuery's event interface, which even allows for the use of delegated events:

```
Widget.prototype.show = function() {
  var event = jQuery.Event("widget:show");
  this.$container.trigger(event);
  if (event.isDefaultPrevented()) {
    // event handler prevents us from showing
    return this;
  }

  this.options.position(this.$thingie, this.$container);
  this.$thingie.show();
```

```
  return this;
};


// listen for all widget:show events
$(document.body).on('widget:show', function(event) {
  if (Math.random() > 0.5) {
    // prevent widget from showing
    event.preventDefault();
  }

  // update widget's data
  $(this).data("last-show", new Date());
});

var widget = new Widget();
widget.show();
```

You can freely choose event names. Avoid using native events[240] for proprietary things and consider namespacing your events. jQuery UI's event names are comprised of the widget's name and the event name `dialogshow`. I find that hard to read and often default to `dialog:show`, mainly because it is immediately clear that this is a custom event, rather than something some browser might have secretly implemented.

Hooks

Traditional getter and setter methods can especially benefit from hooks. Hooks usually differ from callbacks in their number and how they're registered. Where callbacks are usually used on an instance level for a specific task, hooks are usually used on a global level to customize values or dispatch custom actions. To illustrate how hooks can be used, we'll take a peek at jQuery's cssHooks[241]:

```
// define a custom css hook
jQuery.cssHooks.custombox = {
  get: function(elem, computed, extra) {
    return $.css(elem, 'borderRadius') == "50%"
```

240. https://developer.mozilla.org/en-US/docs/DOM/DOM_event_reference
241. http://api.jquery.com/jQuery.cssHooks/

```
        ? "circle"
        : "box";
    },
    set: function(elem, value) {
      elem.style.borderRadius = value == "circle"
        ? "50%"
        : "0";
    }
};

// have .css() use that hook
$("#some-selector").css("custombox", "circle");
```

By registering the hook `custombox` we've given jQuery's `.css()` method the ability to handle a CSS property it previously couldn't. In my article jQuery hooks[242], I explain the other hooks that jQuery provides and how they can be used in the field. You can provide hooks much like you would handle callbacks:

```
DateInterval.nameHooks = {
  "yesterday" : function() {
    var d = new Date();
    d.setTime(d.getTime() - 86400000);
    d.setHours(0);
    d.setMinutes(0);
    d.setSeconds(0);
    return d;
  }
};

DateInterval.prototype.start = function(date) {
  if (date === undefined) {
    return new Date(this.startDate.getTime());
  }

  if (typeof date === "string" &&
DateInterval.nameHooks[date]) {
    date = DateInterval.nameHooks[date]();
  }
```

242. http://blog.rodneyrehm.de/archives/11-jQuery-Hooks.html

```
  if (!(date instanceof Date)) {
    date = new Date(date);
  }

  this.startDate.setTime(date.getTime());
  return this;
};


var di = new DateInterval();
di.start("yesterday");
```

In a way, hooks are a collection of callbacks designed to handle custom values within your own code. With hooks you can stay in control of almost everything, while still giving API users the option to customize.

Generating Accessors

Any API is likely to have multiple accessor methods (getters, setters, executors) doing similar work. Coming back to our `DateInterval` example, we're most likely providing `start()` and `end()` to allow manipulation of intervals. A simple solution could look like:

```
DateInterval.prototype.start = function(date) {
  if (date === undefined) {
    return new Date(this.startDate.getTime());
  }
```

```
  this.startDate.setTime(date.getTime());
  return this;
};

DateInterval.prototype.end = function(date) {
  if (date === undefined) {
    return new Date(this.endDate.getTime());
  }

  this.endDate.setTime(date.getTime());
  return this;
};
```

As you can see we have a lot of repeating code. A DRY (Don't Repeat Yourself) solution might use this generator pattern:

```
var accessors = ["start", "end"];
for (var i = 0, length = accessors.length; i < length;
i++) {
  var key = accessors[i];
  DateInterval.prototype[key] = generateAccessor(key);
}

function generateAccessor(key) {
  var value = key + "Date";
  return function(date) {
    if (date === undefined) {
      return new Date(this[value].getTime());
    }

    this[value].setTime(date.getTime());
    return this;
  };
}
```

This approach allows you to generate multiple similar accessor methods, rather than defining every method separately. If your accessor methods require more data to setup than just a simple string, consider something along the lines of:

```
var accessors = {"start" : {color: "green"}, "end" :
{color: "red"}};
```

```
for (var key in accessors) {
  DateInterval.prototype[key] = generateAccessor(key,
accessors[key]);
}

function generateAccessor(key, accessor) {
  var value = key + "Date";
  return function(date) {
    // setting something up
    // using `key` and `accessor.color`
  };
}
```

In the chapter *Handling Arguments* we talked about a method pattern to allow your getters and setters to accept various useful types like maps and arrays. The method pattern itself is a pretty generic thing and could easily be turned into a generator:

```
function wrapFlexibleAccessor(get, set) {
  return function(name, value) {
    var map;

    if (jQuery.isPlainObject(name)) {
      // setting a map
      map = name;
    } else if (value !== undefined) {
      // setting a value (on possibly multiple names),
convert to map
      keys = name.split(" ");
      map = {};
      for (var i = 0, length = keys.length; i < length;
i++) {
        map[keys[i]] = value;
      }
    } else {
      return get.call(this, name);
    }

    for (var key in map) {
      set.call(this, name, map[key]);
    }
```

```
    return this;
  };
}

DateInterval.prototype.values = wrapFlexibleAccessor(
  function(name) {
    return name !== undefined
      ? this.values[name]
      : this.values;
  },
  function(name, value) {
    this.values[name] = value;
  }
);
```

Digging into the art of writing DRY code is well beyond this chapter. Rebecca Murphey[243] wrote Patterns for DRY-er JavaScript[244] and Mathias Bynens'[245] slide deck on how DRY impacts JavaScript performance[246] are a good start, if you're new to the topic.

The Reference Horror

Unlike other languages, JavaScript doesn't know the concepts of *pass by reference* nor *pass by value*. Passing data by value is a safe thing. It makes sure data passed to your API and data returned from your API may be modified outside of your API without altering the state within. Passing data by reference is often used to keep memory overhead low, values passed by reference can be changed anywhere outside your API and affect state within.

In JavaScript there is no way to tell if arguments should be passed by reference or value. Primitives (strings, numbers, booleans) are treated as *pass by value*, while objects (any object, including Array, Date) are handled in a way that's comparable to *by reference*. If this is the first you're hearing about this topic, let the following example enlighten you:

243. https://twitter.com/rmurphey
244. http://rmurphey.com/blog/2010/07/12/patterns-for-dry-er-javascript/
245. https://twitter.com/mathias
246. http://slideshare.net/mathiasbynens/how-dry-impacts-javascript-performance-faster-javascript-execution-for-the-lazy-developer

```
// by value
function addOne(num) {
  num = num + 1; // yes, num++; does the same
  return num;
}

var x = 0;
var y = addOne(x);
// x === 0 <--
// y === 1

// by reference
function addOne(obj) {
  obj.num = obj.num + 1;
  return obj;
}

var ox = {num : 0};
var oy = addOne(ox);
// ox.num === 1 <--
// oy.num === 1
```

The *by reference* handling of objects can come back and bite you if you're not careful. Going back to the `DateInterval` example, check out this bugger:

```
var startDate = new Date(2012, 0, 1);
var endDate = new Date(2012, 11, 31)
var interval = new DateInterval(startDate, endDate);
endDate.setMonth(0); // set to january
var days = interval.days(); // got 31 but expected 365 -
ouch!
```

Unless the constructor of DateInterval *made a copy* (`clone` is the technical term for a copy) of the values it received, any changes to the original objects will reflect on the internals of DateInterval. This is *usually* not what we want or expect.

Note that the same is true for values returned from your API. If you simply return an internal object, any changes made to it outside of your API will be reflected on your internal data. This is most certainly not what you want. jQuery.extend()[247], _.extend()[248] and Protoype's Object.extend[249] allow you to easily escape the reference horror.

If this summary did not suffice, read the excellent chapter By Value Versus by Reference[250] from O'Reilly's JavaScript – The Definitive Guide[251].

The Continuation Problem

In a fluent interface, all methods of a chain are executed, regardless of the state that the base object is in. Consider calling a few methods on a jQuery instance that contain no DOM elements:

```
jQuery('.wont-find-anything')
  // executed although there is nothing to execute
against
  .somePlugin().someOtherPlugin();
```

In non-fluent code we could have prevented those functions from being executed:

```
var $elem = jQuery('.wont-find-anything');
if ($elem.length) {
  $elem.somePlugin().someOtherPlugin();
}
```

Whenever we chain methods, we lose the ability to prevent certain things from happening—we can't escape from the chain. As long as the API developer knows that objects can have a state where methods don't actually do anything but `return this;`, everything is fine. Depending on what your methods do internally, it may help to prepend a trivial `is-empty` detection:

```
jQuery.fn.somePlugin = function() {
  if (!this.length) {
    // "abort" since we've got nothing to work with
    return this;
  }

  // do some computational heavy setup tasks
```

247. http://api.jquery.com/jQuery.extend/
248. http://underscorejs.org/#extend
249. http://api.prototypejs.org/language/Object/extend/
250. http://docstore.mik.ua/orelly/webprog/jscript/ch11_02.htm
251. http://docstore.mik.ua/orelly/webprog/jscript/index.htm

```
  for (var i = 10000; i > 0; i--) {
    // I'm just wasting your precious CPU!
    // If you call me often enough, I'll turn
    // your laptop into a rock-melting jet engine
  }

  return this.each(function() {
    // do the actual job
  });
};
```

Handling Errors

I was lying when I said we couldn't escape from the chain—there is an `Exception` to the rule (pardon the pun ☺).

We can always eject by throwing an Error (Exception). Throwing an Error is considered a deliberate abortion of the current flow, most likely because you came into a state that you couldn't recover from. But beware—not all Errors are helping the debugging developer:

```
// jQuery accepts this
$(document.body).on('click', {});

// on click the console screams
//   TypeError: ((p.event.special[l.origType] ||
{}).handle || l.handler).apply is not a function
//   in jQuery.min.js on Line 3
```

Errors like these are a major pain to debug. Don't waste other people's time. Inform an API user if he did something stupid:

```
if (Object.prototype.toString.call(callback) !==
'[object Function]') { // see note
  throw new TypeError("callback is not a function!");
}
```

Note: `typeof callback === "function"` should not be used, as older browsers may report objects to be a `function`, which they are not. In Chrome (up to version 12) `RegExp` is such a case. For convenience, use jQuery.isFunction()[252] or _.isFunction()[253].

Most libraries that I have come across, regardless of language (within the weak-typing domain) don't care about rigorous input validation. To be honest, my own code only validates where I anticipate developers stumbling. Nobody really does it, but all of us should. Programmers are a lazy bunch—we don't write code just for the sake of writing code or for some cause we don't truly believe in. The developers of Perl6 have recognized this being a problem and decided to incorporate something called *Parameter Constraints*. In JavaScript, their approach might look something like this:

```
function validateAllTheThings(a, b {where typeof b ===
"numeric" and b < 10}) {
  // Interpreter should throw an Error if b is
  // not a number or greater than 9
}
```

While the syntax is as ugly as it gets, the idea is to make validation of input a top-level citizen of the language. JavaScript is nowhere near being something like that. That's fine—I couldn't see myself cramming these constraints into the function signature anyways. It's admitting the problem (of weakly-typed languages) that is the interesting part of this story.

JavaScript is neither weak nor inferior, we just have to work a bit harder to make our stuff really robust. Making code robust does not mean accepting any data, waving your wand and getting some result. Being robust means not accepting rubbish *and telling the developer about it*.

252. http://api.jquery.com/jQuery.isfunction/
253. http://underscorejs.org/#isFunction

Think of input validation this way: A couple of lines of code behind your API can make sure that no developer has to spend hours chasing down weird bugs because they accidentally gave your code a string instead of a number. This is the one time you can tell people *they're wrong* and they'll actually love you for doing so.

Going Asynchronous

So far we've only looked at synchronous APIs. Asynchronous methods usually accept a callback function to inform the outside world, once a certain task is finished. This doesn't fit too nicely into our fluent interface scheme, though:

```
Api.protoype.async = function(callback) {
  console.log("async()");
  // do something asynchronous
  window.setTimeout(callback, 500);
  return this;
};
Api.protoype.method = function() {
  console.log("method()");
  return this;
};


// running things
api.async(function() {
  console.log('callback()');
}).method();

// prints: async(), method(), callback()
```

This example illustrates how the asynchronous method `async()` begins its work but immediately returns, leading to `method()` being invoked before the actual task of `async()` completed. There are times when we want this to happen, but generally we expect `method()` to execute *after* `async()` has completed its job.

DEFERREDS (PROMISES)

To some extent we can counter the mess that is a mix of asynchronous and synchronous API calls with Promises[254]. jQuery knows them as Deferreds[255]. A Deferred is returned in place of your regular `this`, which forces you to eject from method chaining. This may seem odd at first,

but it effectively prevents you from continuing synchronously after invoking an asynchronous method:

```
Api.protoype.async = function() {
  var deferred = $.Deferred();
  console.log("async()");

  window.setTimeout(function() {
    // do something asynchronous
    deferred.resolve("some-data");
  }, 500);

  return deferred.promise();
};


api.async().done(function(data) {
  console.log("callback()");
  api.method();
});

// prints: async(), callback(), method()
```

The Deferred object let's you register handlers using `.done()`, `.fail()`, `.always()` to be called when the asynchronous task has completed, failed, or regardless of its state. See Promise Pipelines In JavaScript[256] for a more detailed introduction to Deferreds.

Debugging Fluent Interfaces

While *Fluent Interfaces* are much nicer to develop with, they do come with certain limitations regarding de-buggability.

As with any code, *Test Driven Development* (TDD) is an easy way to reduce debugging needs. Having written URI.js in TDD, I have not come across major pains regarding debugging my code. However, TDD only *reduces* the need for debugging—it doesn't replace it entirely.

Some voices on the internet suggest writing out each component of a chain in their separate lines to get proper line-numbers for errors in a stack trace:

254. http://wiki.commonjs.org/wiki/Promises/A
255. http://api.jquery.com/category/deferred-object/
256. http://sitr.us/2012/07/31/promise-pipelines-in-javascript.html

```
foobar.bar()
  .baz()
  .bam()
  .someError();
```

This technique does have its benefits (though better debugging is not a solid part of it). Code that is written like the above example is even simpler to read. Line-based differentials (used in version control systems like SVN, GIT) might see a slight win as well. Debugging-wise, it is only Chrome (at the moment), that will show `someError()` to be on line four, while other browsers treat it as line one.

Adding a simple method to logging your objects can already help a lot—although that is considered "manual debugging" and may be frowned upon by people used to "real" debuggers:

```
DateInterval.prototype.explain = function() {
  // log the current state to the console
  console.dir(this);
};

var days = (new Date(2012, 0, 1))
  .until(2012, 11, 31) // returns DateInterval instance
  .explain() // write some infos to the console
  .days(); // 365
```

FUNCTION NAMES

Throughout this chapter you've seen a lot of demo code in the style of `Foo.prototype.something = function(){}`. This style was chosen to keep examples brief. When writing APIs you might want to consider either of the following approaches, to have your console properly identify function names:

```
Foo.prototype.something = function something() {
  // yadda yadda
};


Foo.prototype.something = function() {
  // yadda yadda
};
Foo.prototype.something.displayName = "Foo.something";
```

The second option `displayName` was introduced by WebKit and later adopted by Firebug / Firefox. `displayName` is a bit more code to write out, but allows arbitrary names, including a namespace or associated object. Either of these approaches can help with anonymous functions quite a bit.

Read more on this topic in Named function expressions demystified[257] by kangax[258].

Documenting APIs

One of the hardest tasks of software development is documenting things. Practically everyone hates doing it, yet everybody laments about bad or missing documentation of the tools they need to use. There is a wide range of tools that supposedly help and automate documenting your code:

- YUIDoc[259] (requires Node.js, npm)
- JsDoc Toolkit[260] (requires Node.js, npm)
- Markdox[261] (requires Node.js, npm)
- Dox[262] (requires Node.js, npm)
- Docco[263] (requires Node.js, Python, CoffeeScript)
- JSDuck[264] (reqires Ruby, gem)
- JSDoc 3[265] (requires Java)

At one point or another all of these tool won't fail to disappoint. JavaScript is a very dynamic language and thus particularly diverse in expression. This makes a lot of things extremely difficult for these tools. The following list features a couple of reasons why I've decided to prepare documentation in vanilla HTML, markdown or DocBoock[266] (if the project is large enough). jQuery, for example, has the same issues and doesn't document their APIs within their code at all.

257. http://kangax.github.com/nfe/
258. https://twitter.com/kangax
259. http://yui.github.com/yuidoc/
260. https://github.com/p120ph37/node-jsdoc-toolkit
261. https://github.com/cbou/markdox
262. https://github.com/visionmedia/dox
263. http://jashkenas.github.com/docco/
264. https://github.com/senchalabs/jsduck
265. https://github.com/jsdoc3/jsdoc
266. http://en.wikipedia.org/wiki/DocBook

1. Function signatures aren't the only documentation you need, but most tools focus only on them.
2. Example code goes a long way in explaining how something works. Regular API docs usually fail to illustrate that with a fair trade-off.
3. API docs usually fail horribly at explaining things *behind the scenes* (flow, events, etc).
4. Documenting methods with multiple signatures is usually a real pain.
5. Documenting methods using option objects is often not a trivial task.
6. Generated Methods aren't easily documented, neither are default callbacks.

If you can't (or don't) want to adjust your code to fit one of the listed documentation tools, projects like Document-Bootstrap[267] might save you some time setting up your home brew documentation.

Make sure your Documentation is more than just some generated API doc. Your users will appreciate any examples you provide. Tell them how your software flows and which events are involved when doing something. Draw them a map, if it helps their understanding of whatever it is your software is doing. And above all: keep your docs in sync with your code!

SELF-EXPLANATORY CODE

Providing good documentation will not keep developers from actually reading your code—your code is a piece of documentation itself. Whenever the documentation doesn't suffice (and every documentation has its limits), developers fall back to reading the actual source to get their questions answered. Actually, you are one of them as well. You are most likely reading your own code again and again, with weeks, months or even years in between.

You should be writing code that explains itself. Most of the time this is a non-issue, as it only involves you thinking harder about naming things (functions, variables, etc) and sticking to a core concept. If you find yourself writing code comments to document how your code does something, you're most likely wasting time—your time, and the reader's as well. Comment on your code to explain *why* you solved the problem this particular way, rather than explaining *how* you solved the problem. The *how* should become apparent through your code, so don't

267. http://gregfranko.com/Document-Bootstrap/

repeat yourself. Note that using comments to mark sections within your code or to explain general concepts is totally acceptable.

Conclusion

- An API is a contract between you (the provider) and the user (the consumer). Don't just change things between versions.
- You should invest as much time into the question *How will people use my software?* as you have put into *How does my software work internally?*
- With a couple of simple tricks you can greatly reduce the developer's efforts (in terms of the lines of code).
- Handle invalid input as early as possible—throw Errors.
- Good APIs are flexible, better APIs don't let you make mistakes.

Continue with Reusable Code for good or for awesome[268] (slides[269]), a Talk by Jake Archibald[270] on designing APIs. Back in 2007 Joshua Bloch gave the presentation How to Design A Good API and Why it Matters[271] at Google Tech Talks. While his talk did not focus on JavaScript, the basic principles that he explained still apply.

Now that you're up to speed on designing APIs, have a look at Essential JS Design Patterns[272] by Addy Osmani[273] to learn more about how to structure your internal code.

Thanks go out to @bassistance[274], @addyosmani[275] and @hellokahlil[276] for taking the time to proof this chapter. ❧

268. http://vimeo.com/35689836
269. http://www.slideshare.net/jaffathecake/reusable-code-for-good-or-for-awesome
270. https://twitter.com/jaffathecake
271. http://www.youtube.com/watch?v=heh4OeB9A-c
272. http://addyosmani.com/resources/essentialjsdesignpatterns/book/
273. https://twitter.com/addyosmani
274. https://twitter.com/bassistance
275. https://twitter.com/addyosmani
276. https://twitter.com/hellokahlil

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' and 'Developing Backbone Applications', having also contributed to open-source projects like Modernizr and jQuery. He is currently working on 'Yeoman' - an opinionated workflow for building beautiful applications.

Christian Heilmann

Christian Heilmann is an international Developer Evangelist working for Mozilla in the lovely town of London, England.

Jörn Zaefferer

Jörn is a freelance web developer, consultant and trainer, residing in Cologne, Germany. Jörn evolved jQuery's testsuite into QUnit, a JavaScript unit testing framework, and maintains it. He created and maintains a number of popular plugins. As a jQuery UI development lead, he focuses on the development of new plugins, widgets and utilities.

Philip Tellis

Philip Tellis is a geek, speedfreak and co-founder of lognormal.com[277] where he tries to infer various characteristics of web users' networks. When he isn't at his computer, he reads, cooks, bikes or falls off his skateboard.

Rodney Rehm

Rod is a Web Developer based in Lottstetten, Germany. He is mainly developing in PHP and JavaScript. He's quite decent with databases and distributed systems for scalability and redundancy. Created URI.js, jQuery Context Menu, Apple's PList in PHP and works on Smarty - PHP Template Engine.

---

277. http://lognormal.com/

Zack Grossbart

Zack Grossbart is an engineer[278], designer, and author[279]. He's a founding member of the Spiffy UI[280] project, the architect of the WordPress Editorial Calendar[281], and an Architecting Engineer and Human Factors Specialist with NetIQ[282]. Zack began loading DOS from a floppy disk when he was five years old. He first worked professionally with computers when he was 15 and started his first software company when he was 16. Zack lives in Cambridge, Massachusetts with his wife and daughter.

278. http://www.zackgrossbart.com/hackito/
279. http://www.zackgrossbart.com/blog/toc/
280. http://www.spiffyui.org/
281. http://stresslimitdesign.com/editorial-calendar-plugin
282. http://www.netiq.com/

About Smashing Magazine

Smashing Magazine[283] 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[284].

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.

About Smashing Media GmbH

Smashing Media GmbH[285] is one of the world's leading online publishing companies in the field of Web design. Founded in 2009 by Sven Lennartz and Vitaly Friedman, the company's headquarters is situated in southern Germany, in the sunny city of Freiburg im Breisgau. Smashing Media's lead publication, Smashing Magazine, has gained worldwide attention since its emergence back in 2006, and is supported by the vast, global Smashing community and readership. Smashing Magazine had proven to be a trustworthy online source containing high quality articles on progressive design and coding techniques as well as recent developments in the Web design industry.

283. http://www.smashingmagazine.com
284. http://www.smashingmagazine.com/publishing-policy/
285. http://www.smashing-media.com