CLAREMONT MCKENNA COLLEGE

JAVASCRIPT AND POLITICS:
HOW A TOY LANGUAGE TOOK OVER THE WORLD

SUBMITTED TO

PROFESSOR ARTHUR LEE

AND

DEAN NICHOLAS WARNER

BY

SEAN MCQUEEN

FOR

SENIOR THESIS

FALL 2013

DECEMBER 2, 2013

Acknowledgements.

To Professor Bull, for his indispensable guidance through my final semester
To my parents, for their support, mentorship and wisdom in all aspects of life and work
To JavaScript developers, for fighting the good fight

Abstract.

The most important programming languages are the ones that manage to capitalize on emerging frontiers in computing. Although JavaScript started its life as a toy language, the explosive growth of the web since 1995 and the invention of the web application have transformed the language's syntax, potential and importance. JavaScript today is powerful and expressive. But is the language good enough to power the future of the web? How does the messy political past of JavaScript affect web development today, and how will it affect web development in the future?

The paper first examines the political history of JavaScript from its origins at Netscape through today. Then a case study of pure JavaScript web development using the NodeJS and AngularJS is presented and analyzed. Finally, several potential paths forward for the language are considered, including a discussion and analysis of Microsoft's TypeScript, Mozilla's ASM.js and Google's Dart.

Table of Contents

JavaScript and Politics

> *We build our computer systems the way we build our cities: over time, without a plan, on top of ruins.* – Ellen Ullman, Author and Programmer

The most important programming languages are the ones that manage to capitalize on emerging frontiers in computing. C rode the rise of open operating systems, Lisp rode the rise of artificial intelligence research and Java rode the rise of business logic applications. But which caused which? Did C become popular because Unix did? Or was Unix so successful because C was so ground-breaking? Like many questions relating to human behavior, it is difficult to tease out causation from correlation. And ultimately these questions are not that interesting. C and Unix are hopelessly intertwined: C was designed to write Unix and Unix was designed to be written in C. Influential tools and their defining creations are often this way. The tool and the creation frame problems through the same lens. And because they happen to provide the right lens at the right time, their fates become tangled.

There are 2.7 billion Internet users in the world and hundreds of millions of websites.[1] Both of those figures have experienced enormous growth since the browser scripting language JavaScript was introduced in 1995. In the same way that the history of C is tied to Unix, JavaScript is inextricably linked to the web. More pages, more content and more Internet users have meant more problems JavaScript was designed to solve. What has been good for the web has been good for JavaScript; and what has been good for JavaScript has been good for the web. The last decade has been extremely good to

[1] "Internet Growth Statistics." *Internet World Stats*. N.p., n.d. Web. 27 Nov. 2013.

both.

Although it is difficult to quantify how many lines of JavaScript are written or how many people are writing them, JavaScript consistently ranks near the top of modern programming language popularity metrics.[2] More important however, than how many people are writing JavaScript, is how many users are running JavaScript programs. And the answer is: nearly everyone. In 2013, the number of mobile Internet devices in use is expected to exceed the number of people in the world.[3] Many of these new devices will run an operating system written in a variant of C, many will support applications written in Java, but for all intents and purposes, *all* of these devices will ship with a JavaScript interpreter. In the browser, JavaScript is king. And on the personal device, the browser is increasingly king.

Yet strangely, for all its popularity, the history of JavaScript is surrounded by conflict. Some JavaScript knowledge is required to write even basic content for the web, meaning that many people's first experience with programming is in JavaScript and a massive amount of JavaScript code is written each day by novices. At the same time, JavaScript experts are pushing the language far beyond what its creators imagined was possible. It has grown to become the language of amateur coders *and* professional software engineers. JavaScript is at home in the copy and paste culture of high-school blogs, and in the highest levels of academic Computer Science. Yet in spite of its popularity, it is often criticized and dismissed. Is JavaScript a toy language? Or is it the

---

[2] "The RedMonk Programming Language Rankings: January 2013." *RedMonk*. N.p., n.d. Web. 27 Nov. 2013.

[3] Arthur, Charles. "Mobile Internet Devices Will Outnumber Humans This Year." *The Guardian*. N.p., 7 Feb. 2013. Web. 27 Nov. 2013. <http://www.theguardian.com/technology/2013/feb/07/mobile-internet-outnumber-people>.

language of the future?

Even five years ago, the answers to these questions were obvious: JavaScript was a toy language on a toy platform. The web was built for viewing pages of static content: Hypertext Markup Language (HTML) is a tool for describing static documents and Hypertext Transfer Protocol (HTTP) is a protocol for fetching those documents. The web was not designed for interacting with data and performing heavy computations. Today, however, the answers are murkier. Clever engineers have pushed the web towards loftier goals. Developers now ask HTML to describe interactive tools and HTTP to manage the input and output of complex programs. JavaScript, a language designed to handle browser events and tie together HTML components, has been pushed the furthest: it is now frequently called upon to define the behavior of complex applications.

The web is no longer just a series of documents strung together with links. The web today is a software platform—perhaps even the beginnings of a distributed operating system. Companies are developing real applications for the web, and web technologies are creeping into desktop operating systems and mobile phones. Programmers are writing JavaScript as if it were C++ and surprisingly, though admittedly with the help of some clever hacks, the language has largely stood the tests of these new responsibilities. As it turns out, the core of the language holds the potential to be flexible and expressive.

The accidental king.

The story of JavaScript's ascendance is somewhat surprising. Just as the web is not owned and operated by any single entity making considered plans, the option to use and extend JavaScript has come from a messy collection of decisions made and remade

daily by thousands of developers. Like the web itself, JavaScript has evolved organically and chaotically; political forces with various and often conflicting agendas have shaped the history, community and syntax of the language. Only hindsight reveals why JavaScript has risen to its dominant position. JavaScript, the most important programming language of the modern era, is the accidental king.

The future of the web and the future of JavaScript are intrinsically tangled. And for millions of web developers—as well as for companies like Google, Amazon and Microsoft—the question of what tools will be used for web application development in the coming years could not be more important. Some have argued that web technologies need to be replaced. JavaScript should be thrown out and we should start from scratch with a more careful design. Others have argued JavaScript is the right language to move forward with, but it needs to be fixed in a big way. Still others see no problem with the current trajectory of the web: maybe we should accept that the requirements of these technologies are impossible to predict or control, and let them grow and change on their own. It is not clear which argument is right, or how these voices will shape the language. What is clear is that the possibilities of the web as a software platform rest on the future of JavaScript.

Browser Wars and Mocha

> *Most languages die in obscurity. Only a few are able to build a following beyond a single project or company. And only a very small number of languages become important.* – Douglas Crockford

At 1:30 AM the morning of October 1, 1997, a group of Microsoft employees placed a giant metal Internet Explorer logo on the lawn of Netscape Communications in Mountain View, California.[4] According to a Reuters report of the incident, the Microsoft employees were returning from an Internet Explorer 4 release party in San Francisco; a card attached to the oversized prop read, "From the IE team."[5]

"It seems awfully immature to resort to fraternity tactics to draw attention," a Netscape spokeswoman said at the time. "We're winning the battle. It's something you'd expect from a startup, not the largest software company in the world."[6] Although this was the company's official position, Netscape employees could not resist using similar tactics in response. By sunrise, the logo was defaced with spray paint and Netscape's foam mascot, a creature named Mozilla, was standing on the wreckage with a placard referring to recent market share percentages: "Netscape 72, Microsoft 18."[7] The Browser Wars, whose winner would bring the web to the masses, were in full swing.

Less than two years earlier, over the course of ten days in May 1995, a software engineer at Netscape named Brendan Eich wrote a browser scripting language called Mocha. Eich is not quite sure which ten days it was. "From a calendar, I think it might

---

[4] "Microsoft Plays Prank on Netscape after Bash." *Mozilla Stomps IE*. N.p., 2 Oct. 1997. Web. 27 Nov. 2013. <http://home.snafu.de/tilman/mozilla/stomps.html>.
[5] Ibid.
[6] Ibid.
[7] Ibid.

have been May 6-15, 1995," he writes in a 2013 answer to a question about Mocha on an online forum. His office-mate at the time might remember, he adds before apologizing, "That's the best I can do."[8]

The Browser Wars.

At the time, those ten days in May were not any different from any other ten days at Netscape in 1995. The mood at the time was one of urgency and excitement, perhaps bordering on mania. Netscape, founded in 1994, was riding the beginning of the "dot-com" boom with their hit product Navigator. Netscape had been born out of a research project at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign. The research project, Mosaic, with development led by 22-year-old Marc Andreessen, was a tool for viewing Internet content through a graphical interface. Although not the first web browser, Mosaic's friendly interface made it immediately popular. Seeing the potential for the project's commercial success, Andreessen left Illinois for Silicon Valley, where he met with Jim Clark, a Stanford professor who had made a fortune as a co-founder of Silicon Graphics. Together, Clark and Andreessen started Netscape.

Web browsers at the time were a brand new way to interact with the Internet. Before open web technologies like HTTP and HTML, graphical interfaces for viewing Internet content had exclusively been closed proprietary systems, like America Online,

[8] Eich, Brendan. "JavaScript: In Which 10 Days of May Did Brendan Eich Write JavaScript (Mocha) in 1995?" *Quora*. N.p., 16 Feb. 2013. Web. 27 Nov. 2013. <http://www.quora.com/JavaScript/In-which-10-days-of-May-did-Brendan-Eich-write-JavaScript-Mocha-in-1995>.

which only allowed access to their own content. And before the graphical interface, accessing content on the Internet was a text-heavy activity reserved for geeks and scientists. The World Wide Web, built on HTTP and HTML, was graphical *and* open, allowing anyone to create content for anyone else to view. The promise of this technology, with browser products like Navigator paving the way for its use in homes and businesses, was enormous.

On August 9, 1995, less than a year after its founding, Netscape offered shares to the public for $28. By the end of that day, shares were trading for $75.[9] Clearly, the web hit a nerve with the public. Netscape was the fastest growing software company in history. Andreessen sums up the feeling in a 2000 interview with Wired Magazine: “In 1995, from Q1 to Q4, Netscape’s revenue went from $5 million to $10 million to $20 million to $40 million. No one had ever seen anything like that... Shit! That’s it! We are a hit!”[10]

Unfortunately for Netscape, celebration would have to be short-lived. Competition was heating up. Seeing the potential of Andreessen’s browser, Microsoft had offered to buy Netscape in 1994.[11] Netscape executives had scoffed at the deal, which they thought was for far too low a sum. At the time, the Redmond software giant, with Bill Gates at the helm, was the gorilla in the room; the company had popularized personal computing. But the explosive growth at Netscape was worrisome for Microsoft. Netscape was calling their product a “distributed operating system,” a claim that threatened

---

[9] Niiler, Eric. "Netscape's IPO Anniversary and the Internet Boom." *Digital Life*. NPR. 9 Aug. 2005. Radio.

[10] Sheff, David. "Wired 8.08: Crank It Up." *Wired.com*. Conde Nast Digital, Aug. 2000. Web. 27 Nov. 2013. <http://www.wired.com/wired/archive/8.08/loudcloud.html?pg=1>.

[11] "Fluent 2012: Brendan Eich, "JavaScript at 17"" *YouTube*. YouTube, 30 May 2012. Web. 27 Nov. 2013. <http://www.youtube.com/watch?v=Rj49rmc01Hs>.

Microsoft's core business. Microsoft knew that it needed its own browser. So in early 1995 the company licensed Mosaic, the same research product that Andreessen had worked on, from Spyglass Inc., the commercial offshoot of the NCSA.[12] By the summer of 1995, Microsoft was only months away from releasing what amounted to a thin varnish over the Mosaic codebase called Internet Explorer.[13] Although late to the browser game, Microsoft was serious about the web. Thanks to the company's success with their Windows operating system, Microsoft had amassed an elite team of software engineers and nearly bottomless resources. Netscape's competition was about to get supercharged, and everyone knew it.

Proving Mocha.

Mocha, Eich's ten-day language, was one of an array of strategic plays that Netscape was making to stay ahead of Microsoft with their Navigator 2.0 release. Given that web content in 1995 was completely static, the need for a HTML scripting language was not obvious. But there were visionaries at Netscape that understood that such a technology was an opportunity for a strategic win. The development of Mocha was mandated from the top. "The impetus was the belief on the part of at least Marc Andreessen and myself, along with Bill Joy of Sun, that HTML needed a 'scripting language,'" said Brendan Eich in a 2008 interview with ComputerWorld.[14] "We aimed to

---

[12] Sink, Eric. "Memoirs From the Browser Wars." Web log post. *Eric Sink*. N.p., Apr. 2003. Web. 27 Nov. 2013. <http://www.ericsink.com/Browser_Wars.html>. =

[13] Ibid.

[14] Hamilton, Naomi. "The A-Z of Programming Languages: JavaScript." *Computerworld*. N.p., 31 July 2008. Web. 27 Nov. 2013. <http://www.computerworld.com.au/article/255293/a-z_programming_languages_javascript/>.

provide a ‘glue language’ for the web designers and part time programmers.”[15]

But even with support from Andreessen, not everyone at Netscape was convinced that Mocha should be a priority. Even if a browser language was important, why create something new? At the time, the Java programming language was gaining popularity and many thought it was a logical language for the browser. In fact, Netscape was in the midst of crafting a deal with Sun Microsystems to support Java in Navigator. So why was Mocha worth the effort? Why not just Java?

This was a reasonable question to ask. But Eich (and Andreessen) thought that that the two languages were different enough that both were needed. The professional software engineers, who would also be coding the backend servers powering websites, deserved a robust and extendable language. On the other hand, the front-end amateurs needed something expressive and easy. In order to appeal to both audiences, a move that could cement Netscape’s market position, Navigator could support Java *and* Mocha. But in order to prove this idea to the company, Eich had to move quickly. Eich wrote the Mocha interpreter in ten days because the fate of Mocha was not sealed. “Doing this work so fast was important,” Eich recalls in a talk at a conference on JavaScript in 2012. “I had to [write the interpreter] for an internal demo, because otherwise people would doubt that it was either real or necessary.”[16] Apparently the Mocha demo was convincing. The language was shipped with Navigator 2.0 in December 1995.

---

[15] Ibid.
[16] "Fluent 2012: Brendan Eich, "JavaScript at 17""

Sun and Netscape.

By the time Navigator 2.0 shipped, Eich's language had been renamed twice. First to "LiveScript" in September 1995, and then in December to "JavaScript," in a licensing deal with Sun, that was aggressively marketing its Java platform at the time. Sun and Netscape had been working together on a major deal to put Java into Navigator. But Sun did not like the idea of Mocha. In their minds, Java was going to be the web language, end of story. Why was Netscape trying to put two languages into Navigator? Andreessen believed strongly in Mocha and was not willing to compromise by leaving it out of Navigator. So in order to satisfy Sun, Netscape agreed to rename the language "JavaScript" as part of the deal.

Over the years, the marketing decision behind the name "JavaScript" has understandably led to a lot of confusion. Because Java was so well regarded at the time, the confusion was probably a good thing for the language in JavaScript's early years. But even today many assume that the two languages are related. Oracle, the company that current owns and develops the Java platform, even includes a webpage dedicated to clearing up the misunderstanding.[17] In spite of some of their shared history, Java and JavaScript are fundamentally different. Although they share some syntactical similarities, the cores of the languages have almost nothing to do with each other. That said, the politics of Java directly impacted JavaScript, before, during and after JavaScript's development.

[17] "How Is JavaScript Different from Java?" *Java*. Oracle, n.d. Web. 27 Nov. 2013. <http://www.java.com/en/download/faq/java_javascript.xml>.

Oak.

Java started its life in 1991 as a language called Oak, written for a research project at Sun that was removed from the rest of the company. According to a Sun document recounting the history of Java written on the third anniversary of Java's public release, Sun's secret "Green Team," with 13 total members, was formed to anticipate and plan for the next big advances in computing. "Their initial conclusion was that at least one significant trend would be the convergence of digitally-controlled consumer devices and computers."[18] With this thesis in mind, the Green Team set out to build technology for consumer devices, mainly televisions. Almost immediately, they realized that such a project demanded that they had to work with many different processor architectures and that C++ was not the best language for the job. Out of frustration, James Gosling, a Green Team software engineer, wrote Oak.[19]

Oak was compiled into a processor-agnostic set of instructions, which then could be run on a "virtual machine" on a number of different machine architectures.[20] As such, Oak allowed Green Team engineers to compile a program once and then run that program on many different devices. In retrospect, this was a major step forward for programming abstraction, but at the time, Oak did not seem critical for Sun. For many years, Oak existed only in a strange back corner of the company, where the Green Team (eventually spun off as a subsidiary called FirstPerson) continued their work with media

---

[18] Byous, Jon. "JAVA TECHNOLOGY: THE EARLY YEARS." Sun Microsystems, 1998. Web. 27 Nov. 2013. <http://web.archive.org/web/20050420081440/http://java.sun.com/features/1998/05/birthday.html>.

[19] Kohlbrenner, Eric. "The History of Virtual Machines." *Core of Information Technology*. N.p., n.d. Web. 25 Nov. 2013. <http://www.cs.gmu.edu/cne/itcore/virtualmachine/history.htm>.

[20] Byous.

technology.[21]

But in 1995, the world was changing rapidly. It became clear to Sun executives that Oak, which was subsequently renamed Java, had some characteristics that made it perfect for programming in the browser. Gosling explained the realization in an interview for the 1998 historical account at Sun: “Even though the [Internet] had been around for 20 years or so, with FTP and telnet, it was difficult to use. Then Mosaic came out in 1993 as an easy-to-use front end to the web, and that revolutionized people’s perceptions. The Internet was being transformed into exactly the network that we had been trying to convince the cable companies they ought to be building.”[22] The processor independent nature of Java, which was not easy to replicate in other serious programming languages at the time, was perfect for browser programs (which had to run on every computer). “It was just an incredible accident. And it was patently obvious that the Internet and Java were a match made in heaven.”[23]

On May 23rd, 1995, John Gage, director of the Science Office at Sun, and Marc Andreessen stepped on the stage at the SunWorld conference to announce simultaneously the public release of Java and the incorporation of Java into Navigator.[24] Although Java had been developed separately from concerns associated with the web, it appeared to the world that Java was built for Navigator. This was exactly what Sun and Netscape executives wanted. The partnership was a strategic play that they knew could advance the momentum of both companies. Together, could they challenge Microsoft? More pessimistically, if they did not stand together against the Microsoft, would they stand a

[21] Ibid.
[22] Ibid.
[23] Ibid.
[24] Ibid.

chance? Douglas Crockford, a major figure in the JavaScript development community, thinks that the deal was a necessity. “At the time there was a lot of excitement about Java and the Netscape browser. Sun and Netscape decided they needed to work together against Microsoft, because if they didn’t join forces, Microsoft would play them off against each other and they’d both lose.”[25]

Java and JavaScript.

The close relationship between Sun and Netscape in 1995 cast a long shadow over the development of JavaScript. According to Brendan Eich, he was hired at Netscape to embed a functional language in Navigator. As Eich writes in a 2008 blog post: “As I’ve often said, and as others at Netscape can confirm, I was recruited to Netscape with the promise of ‘doing Scheme’ in the browser...”[26] But when he arrived at Netscape and the development of the HTML scripting language was about to begin, Eich recalls, management had new requirements. “The diktat from upper engineering management was that the language must ‘look like Java.’”[27] There was a strong desire by marketers at Netscape to associate their browser with Java. With many people predicting that Java was going to take over the world with its “Write Once, Run Everywhere” philosophy, perhaps that desire was well-founded. Regardless of its marketing potential however, asking that the language look like Java, Eich notes, “ruled out Perl, Python, and Tcl, along with

[25] "Quick History of Javascript by Crockford." *YouTube*. YouTube, 05 Sept. 2013. Web. 27 Nov. 2013. <http://www.youtube.com/watch?v=t7_5-XYrkqg>.

[26] Eich, Brendan. "Popularity." Web log post. N.p., 3 Apr. 2008. Web. 27 Nov. 2013. <https://brendaneich.com/2008/04/popularity/>.

[27] Ibid.

Scheme," which otherwise would have been reasonable choices to adapt for Navigator.[28]

In order to satisfy a number of conflicting goals, Eich and his superiors knew that a new language had to be written. And once that became clear, there was pressure to make it like Java, but not copy Java too closely. Eich recounts, "If I had done classes in JavaScript back in May 1995, I would have been told that it was too much like Java or that JavaScript was competing with Java... I was under marketing orders to make it look like Java but not make it too big for its britches... [it] needed to be a silly little brother language."[29]

The good, the bad and the ugly.

Given all that surrounded the origins of JavaScript, and the requirements put on its development, the fact that Eich got anything working might be a miracle. JavaScript had to simultaneously look like Java, avoid copying Java and act very differently than Java. It had to play nice with HTML and Navigator's internals and be interpreted on any machine architecture. Lastly, it had to be designed and written on an extremely short timeline. Although the language was not released until December, Eich, who was the only developer on the project until 1996, had little time to devote to the language after the infamous ten days in May.[30] Mostly this was because he had to make sure the language could be embedded into Navigator, a task that ended up being the bulk of the work. "I spent the rest of 1995 embedding [JavaScript] in the Netscape browser and creating what

---

[28] Ibid.

[29] Severance, Charles. "JavaScript: Designing a Language in 10 Days." *IEEE Computer Society*. N.p., Feb. 2012. Web. 27 Nov. 2013. <http://www.computer.org/csdl/mags/co/2012/02/mco2012020007.html>.

[30] Hamilton.

has become known as the 'DOM': APIs from [JavaScript] to control windows, documents, forms, links, images, etc., and to respond to events and run code from timers," said Eich in the ComputerWorld interview.[31] Without the DOM interface, JavaScript was of little interest to anyone at Netscape. So the initial design of the language was accelerated.

The language that Eich produced had C-like brace syntax and took inspiration from many parts of Java's standard library. But that is largely where the similarities with Java end. JavaScript's functional concepts are inspired by Scheme, and the language's object-oriented concepts are largely inspired by Self and Smalltalk. JavaScript is an impressively coherent language given its origins, but Eich is not a superhero: the stresses of the development period clearly took a toll on the language's design.

Although JavaScript supports some functional concepts like closures and first-class functions, the language is fundamentally object-oriented. However, unlike other modern object-oriented languages, JavaScript does not support classes. JavaScript objects can be polymorphic, and a programmer can encapsulate data within a JavaScript object, but inheritance in JavaScript is not as clean as the class inheritance is in Java or C++. JavaScript's type system is also poorly designed. For instance, there is only one number type in JavaScript.[32] Unlike other modern languages, in which numbers can be represented as integers or floating point numbers for different levels of precision, all numbers in JavaScript are implemented as IEEE floating point numbers. This leads to unfortunate consequences:

---

[31] Ibid.

[32] "ECMAScript Language Specification 5.1." Ecma International, 2011. Web. 27 Nov. 2013. <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>.

```
> 0.1 + 0.2 == 0.30000000000000004
true
```

Dealing with large and small numbers that lack reasonable precision in the IEEE standard is difficult in JavaScript. It is hard to imagine a bank using the language to keep track of interest payments, for example. But that is not where the problems with JavaScript's type system ends.

There are two comparison operators in JavaScript "==" and "===", with different but almost identical meanings. "==" includes automatic type conversion, whereas "===" does not. JavaScript's strange auto-type conversation rules mean that some odd things are true within the language:

```
> '0' == 0  // a equals b?
true
> 0 == ''  // b equals c?
true
> '0' == '' // a equals c?
false
```

Additionally, JavaScript has more than its share of empty types. The values null, undefined, NaN ("not a number"), Infinity and false are all used in different contexts to signify that a variable holds a "bad" value that cannot be operated on in a meaningful way. Each of these has slightly different meaning, but those meanings are not always obvious. Testing equality among these "bad" values can be problematic. To make things extra confusing, NaN does not equal itself:

```
> NaN === NaN
false
> null == undefined
true
> null === undefined
false
```

There is a long list of language design problems that make JavaScript difficult to work with. Among them, JavaScript uses function scope rather than block scope, variables can be declared in the global scope of a program from any scope, the language does not have any built in module system and JavaScript has no built-in concept of input or output.[33] In short: as it was written, JavaScript was not good for writing anything more significant than short scripts. Although Eich's work was impressive given the circumstances, he did not produce a high quality language. JavaScript was severely limited in 1995.

Microsoft crushes Netscape.

On August 13, 1996, Microsoft released Internet Explorer 3. Among a multitude of improvements that the browser brought over its predecessors, Internet Explorer 3 shipped with JavaScript support. Microsoft had reversed engineered JavaScript with an implementation they called JScript (for copyright reasons).[34] JScript was essentially a direct copy of JavaScript. There were small differences, but for the most part, a program written in JavaScript could be run in both browsers. This did not seem like a big deal at

[33] "A Re-introduction to JavaScript." *Mozilla Developer Network*. Mozilla, n.d. Web. 27 Nov. 2013. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript>.

[34] "Microsoft Internet Explorer 3.0 Beta Now Available." *Microsoft News Center*. N.p., 29 May 1996. Web. 27 Nov. 2013. <http://www.microsoft.com/en-us/news/press/1996/may96/ie3btapr.aspx>.

the time, except to Netscape executives (who were obviously not happy about it). But in retrospect, it was this development that took JavaScript from being a proprietary tool to something of a standard: as of 1997, the two largest browsers had committed to supporting the language.

Internet Explorer 3 was replaced by Internet Explorer 4 in late 1997, which was replaced not too much later by IE5, and then IE6. With each iteration, Microsoft took a larger and larger portion of market share away from Netscape. Microsoft had a number of market advantages that made their progress much easier. The most conspicuous was financial resources. Profits from Microsoft's other businesses, $559 million in the second quarter of 1996, allowed Microsoft to give away Internet Explorer for free.[35] Netscape could not afford to compete this aggressively. Microsoft's second big advantage was its domination of the operating system space: Windows faced no real competition. Taking advantage of this, Microsoft began bundling Internet Explorer with Windows 95. Most copies of Windows at the time were being sold to people who had never owned a personal computer before, let alone used a browser. So it is possible that many did not know they *could* download a competing product. As the market for web browsers grew rapidly, and spread to encompass millions of less technical people, Microsoft's bundling strategy made Internet Explorer the default for new users. Soon, the default became the standard. And by 1998, Internet Explorer owned the market. In that year Microsoft was sued for anti-competitive behavior (partially because of its software bundling strategy), but the war was over. By 2002, almost all Internet users were using Internet Explorer.

---

[35] "Windows 95 Opens Door for Microsoft's 50% Profit Jump." *Los Angeles Times*. Los Angeles Times, 23 July 1996. Web. 27 Nov. 2013. <http://articles.latimes.com/1996-07-23/business/fi-27151_1_fourth-quarter-profit>.

Although its original implementation did not become its dominant form, JavaScript lived on. When other browsers came along in the mid-2000s and took market share away from Internet Explorer, implementing JavaScript for the makers of these browsers was a no-brainer. Mozilla introduced Firefox, Apple introduced Safari and Google introduced Chrome. All three of these browsers supported JavaScript from their first commercial release. The competition between Netscape and Microsoft, followed by Internet Explorer's rise to prominence, propelled Eich's language to the status of a standard. At this point, switching to a different scripting language would be nearly impossible: JavaScript is the language of the web.

Evolution and Revolution

> *There are two ages of the Internet – before Mosaic, and after... In twenty-four months, the Web has gone from being unknown to absolutely ubiquitous.* – A Brief History of Cyberspace, 1995

In 1994, two graduate students at Stanford, Jerry Yang and David Filo, published a webpage they called "Jerry and David's Guide to the World Wide Web."[36] At first the page was only a collection of Jerry's golf scores and some links to content the two enjoyed. But as they added more links, almost overnight, the guide became a hit. According to Mental Floss magazine, "[Jerry and David] quickly realized they might need a name that took less than three minutes to say, so they switched to a word they liked from the dictionary – one that described someone who was 'rude, unsophisticated, and uncouth.'"[37] Yahoo! was born.

Yahoo! was the first example of a web portal: a website that provided hierarchical lists of links, allowing users to browse the web from a central hub. When the web was small enough, people could remember off the top of their heads which webpages they might want to visit and what their addresses were. For instance, in December 1991, only a few years earlier, there were only 10 websites.[38] As the web grew however, the need for link aggregators like Yahoo! to provide content organization as a service became important. For a while, this model was extremely successful. Yahoo! captured the public's imagination. Unfortunately for the portal sites however, the growth of the web was

---

[36] Clark, Andrew. "How Jerry's Guide to the World Wide Web Became Yahoo." *The Guardian*. N.p., 1 Feb. 2008. Web. 27 Nov. 2013. <http://www.theguardian.com/business/2008/feb/01/microsoft.technology>.

[37] Trex, Ethan. "9 People, Places & Things That Changed Their Names." *Mental_floss Blog*. N.p., n.d. Web. 27 Nov. 2013. <http://blogs.static.mentalfloss.com/blogs/archives/22707.html>.

[38] *A Brief History of Web Standards*. Digital image. Vitamin T, 30 Oct. 2011. Web. 27 Nov. 2013. <http://visual.ly/brief-history-web-standards>.

unrelenting. Quickly the idea that one group of people could understand the entire web, and manually manage a set of links to the best content became unrealistic. Before long, to keep up with the pace of content growth, Yahoo! would need to add hundreds of links to its lists every day.

In 1996, another pair of Stanford graduate students saw the problem differently. “Each computer was a node, and each link on a web page was a connection between nodes – a classic graph structure,” Larry Page, one of these students, told Wired Magazine in 2008.[39] The web, Page and his partner Sergey Brin theorized, was the largest graph ever created.[40] And it was growing at a breakneck pace. In order to organize this graph so that it could be reasonably navigated by Internet users, they needed a heuristic for a webpage’s importance. Their key insight was that the value of a page was already encoded in the graph: the nodes with the most incoming connections were the most important. Using this heuristic, Page and Brin wrote an algorithm to crawl the web graph and assign scores to webpages. The two built a search engine on top of their results and opened it to the public. The quality of the search results provided by Google, the search engine built around the algorithm, was unmatched.

The web grew so quickly in the late 90s that machine learning techniques, like Google’s, were required to make sense of it, but the web was not finished growing. In 1998, when Google’s search engine first indexed the web, it recorded 28 million unique pages. In 2000, it first recorded over a billion. And in 2008, 10 years after the first crawl,

---

[39] Battelle, John. "Wired 13.08: The Birth of Google." *Wired.com*. Conde Nast Digital, Aug. 2008. Web. 27 Nov. 2013. <http://www.wired.com/wired/archive/13.08/battelle.html>.

[40] Ibid.

Google's announced that their index had recorded over a trillion unique pages.[41]

The number of Internet users moved in tandem with the increase in content published on the web. In December 1995, the month Netscape launched Navigator 2.0, there were estimated to be 16 million Internet users. In 2013, there are 2.7 billion.[42] Such dramatic changes in the usage of the Internet caused major changes in the nature of the technology. Once a province of static information for research scientists, the Internet's open web became a multifaceted platform for almost everything. Today "the web" is a cultural phenomenon. And under the hood of this phenomenon, powering its growth and feeding off its successes, is a funny language called JavaScript.

Growing pains.

Users demanded more webpage interaction as the web became mainstream. Flashing banners, buttons that responded to clicks, and embedded games became popular attractions that drove traffic and ad revenue. Web developers delivered these effects through a number of different technologies, but JavaScript had a number of advantages when it came to making the web interactive. First of all, JavaScript implementation details could not be hidden; because developers could read the JavaScript running on any given page, JavaScript programs on the web were open-source by default. Good JavaScript ideas could travel virally through the Internet. But more importantly, JavaScript had direct access to browser APIs, the HTML it was embedded in and the styling of that HTML. Although JavaScript was arguably much less powerful than Java

---

[41] "We Knew the Web Was Big..." *Google Official Blog*. Google, 25 July 2008. Web. 27 Nov. 2013. <http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html>.

[42] "Internet Growth Statistics."

applets or Flash for describing interaction, it ended up being the best tool for the job.

Unfortunately, many developers disliked writing JavaScript. The major weaknesses of the language's design, coupled with slight differences between the implementations of JavaScript in web browsers (or even different versions of the same browser) meant that writing JavaScript could mean fighting a constant battle. Especially for developers accustomed to languages like C, C++ or Java, writing in JavaScript, which lacked a module system and a reasonable type system, was not ideal. Nevertheless, as the world came online and traffic rewarded webpages with more complexity and interaction, developers had no choice. As the size of the web grew exponentially, so too did developer reliance on JavaScript. Fortunately for web developers, a series of events since 1995 has dramatically improved the experience of developing software in JavaScript.

Standards and fragmentation.

Many of the early problems associated with JavaScript were actually problems with browsers, not problems with the language. In particular, standards fragmentation and the underdevelopment of browsers deserve some blame. In 1994, anticipating a need to maintain standards for the open web and avoid inconsistencies, Tim Berners-Lee and others at CERN founded the World Wide Web Consortium (W3C), a group dedicated to recommending standards for HTML, XML and other building blocks of the web. Despite the best efforts of W3C and other standards bodies however, by 1998 Netscape and Microsoft had each captured about 50% of the browser market and their 4.0 releases were largely incompatible. Fierce competition had pushed the two companies to ignore W3C's recommendations and build proprietary features that only worked in their own browsers.

For instance, Netscape introduced an HTML blink tag (<blink>) that made text flash, a feature that is still not supported by Internet Explorer or recognized by W3C.[43] Browser inconsistencies meant that developing for the web became increasingly difficult in the late 90s.

Luckily, most of the problems resulting from browser inconsistencies were fixed as side-effects of larger trends. As Internet Explorer became the dominant browser, the problems of browser fragmentation became less and less pressing. By the year 2000, even if Microsoft wanted to ignore W3C's recommendations, the mystery of what was supported for which users was mostly gone. In addition to this, Internet Explorer (and its new competitors, when they arrived) matured into better software over time. Fewer bugs, better built-in tools and more consistent JavaScript behavior meant that, without any effort to fix or improve JavaScript in particular, developing in the language improved considerably.

Ecma evolution.

Not all of JavaScript's problems could be blamed on the browsers. As JavaScript's popularity ballooned, there was considerable pressure to address the fundamental design flaws of the language. In November 1996, several months after Microsoft had released Internet Explorer 3 with JScript, Netscape delivered JavaScript to Ecma International for standardization.[44] Douglas Crockford, a well-known JavaScript developer and somewhat

[43] "Blink." *Mozilla Developer Network*. Mozilla, n.d. Web. 27 Nov. 2013. <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/blink>.

[44] Rauschmayer, Axel. *The Past, Present, and Future of JavaScript Where We've Been, Where We Are, and What Lies Ahead*. Sebastopol, CA: O'Reilly Media, 2012. Print. Pg. 5

of a JavaScript historian, notes that Ecma was not Netscape's first choice. "[Netscape] went to W3C and said 'OK, we've got a language for you to standardize.'" [45] But W3C had been upset with Netscape for ignoring browser standards. "W3C had been waiting a long time for an opportunity to tell Netscape to 'go to hell.'"[46] Netscape could not take no for an answer however. JavaScript had to be standardized; Microsoft's adoption of JScript had made it clear that the language was not just Netscape's anymore.

After trying a few more standards boards, Netscape ended up at Ecma, originally founded in 1961 as the European Computer Manufacturers Association. Ecma was, as Crockford notes, "a long way to go for a California software company."[47] Nevertheless, standardization of the language began immediately at Ecma. Unfortunately, because the name JavaScript was licensed exclusively by Sun for Netscape to use, Ecma, like Microsoft, had to choose a different name for the same language. They eventually settled on ECMAScript, a name that Brendan Eich has referred to "an unwanted trade name that sounds like a skin disease."[48] The first version of ECMAScript, which essentially just codified the existing implementation of JavaScript at Netscape, was adopted by the Ecma general assembly in June 1997.[49]

Ecma has improved the language significantly since taking control. The third edition of ECMAScript, adopted in June 1999, was the first edition where Ecma made meaningful changes to the standard. According to the specification document, ECMAScript's third edition adds "powerful regular expressions, better string handling,

---

[45] "Quick History of Javascript by Crockford "
[46] Ibid.
[47] Ibid.
[48] Eich, Brendan. "Will There Be a Suggested File Suffix for Es4?" *Es4-discuss*. Mozilla, 3 Oct. 2006. Web. 27 Nov. 2013. <https://mail.mozilla.org/pipermail/es-discuss/2006-October/000133.html>.
[49] "ECMAScript Language Specification 5.1."

new control statements, try/catch exception handling, tighter definition of errors, and formatting for numeric output."[50] Several years later, ambitious work on ECMAScript 4 was started but abandoned due to arguments about language complexity. Instead, in August 2008, the committee in charge of ECMAScript agreed to introduce a more incremental change to the language followed subsequently with a more major release.[51] ECMAScript 5, the more incremental release, was agreed upon in December 2009.[52] According to the specification document, ECMAScript 5 adds a number of features to the language, but only two stand out. First, ECMAScript 5 added support for the JSON object encoding format, including a built-in JSON object with a "parse" method for turning JSON strings into JavaScript objects. Second, the standard defined an opt-in subset of ECMAScript called "strict mode." Among other requirements, strict mode makes it impossible to assign variables at the global scope.[53] In order to make transition to future version of JavaScript easier on legacy programs, ECMAScript 5's strict mode also reserves some extra words for the language (for instance "let"), even though they are not currently used by the language. As of 2013, all modern browsers implement ECMAScript 5.[54]

Although Ecma's changes have improved the core of the language significantly, the true forces pushing JavaScript development forward have been outside the standards

---

[50] "ECMAScript Language Specification 3. "

[51] Rauschmayer. Pg. 10

[52] Rauschmayer, Axel. "A Brief History of ECMAScript Versions (including Harmony/ES.next)." *Web Builder Zone*. N.p., Mar. 2011. Web. 27 Nov. 2013. <http://css.dzone.com/articles/brief-history-ecmascript>.

[53] "Strict Mode." *Mozilla Developer Network*. N.p., n.d. Web. 27 Nov. 2013. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/Strict_mode>.

[54] "JavaScript." *Mozilla Developer Network*. N.p., n.d. Web. 26 Nov. 2013. <https://developer.mozilla.org/en-US/docs/Web/JavaScript>.

board. Clever engineering and dramatic shifts in the best practices for JavaScript development have allowed developers to work *around* problems with JavaScript, to develop real software in the language.

HTML and HTTP.

Tim Berners-Lee, a British computer scientist, is credited with inventing the World Wide Web, the set of technologies built on the Internet for creating, sharing and viewing text and images. While working in at the European physics laboratory CERN, which in the late 1980s was the largest Internet node, Berners-Lee had a vision of a connected system for graphical information sharing.[55] Although most of the components of the web already existed (for instance the Internet, Hypertext and multi-font text objects), no one at the time saw the larger potential of these components. “It was a step of generalizing, going to a higher level of abstraction, thinking about all the documentation systems out there as being possibly part of a larger imaginary documentation system,” Berners-Lee said in 2007 interview. He and his team created the Hypertext Transfer Protocol (HTTP) and Berners-Lee himself wrote the first web client and server in 1990.[56]

As originally designed, HTML is an excellent tool for describing text documents and HTTP is an excellent tool for retrieving those documents. These technologies work together well for what they were designed to do, and they solved Berners-Lee’s immediate problems at CERN. However, in retrospect, they are limited. Berners-Lee

---

[55] "Sir Timothy Berners-Lee Interview." *Academy of Achievement*. N.p., 22 July 2007. Web. 27 Nov. 2013. <http://www.achievement.org/autodoc/page/ber1int-1http://www.achievement.org/autodoc/page/ber1int-1>.

[56] "Internet Hall of Fame Innovator: Tim Berners-Lee." *Internet Hall of Fame*. N.p., n.d. Web. 27 Nov. 2013. <http://internethalloffame.org/inductees/tim-berners-lee>.

could not have anticipated how far web developers would want to push his technologies. As evidence of this, the first working version of HTTP only had one method ("GET") that always returned an HTML page.[57] And although subsequent version of HTTP, including the version that is used today, included more complex behavior (for example a "POST" method for sending data to a server), it is clear that the protocol was designed for a very simple model of client and server interaction. In Berners-Lee's original model for the web, a client asked for a particular page, and the server sent it back, period.

This simple model for client and server interaction was quickly expanded to fit the growing needs of the web. For instance, a web server did not need to return the same page every time it received a request, like the original web servers at CERN did. Instead, a server-side program could manage application logic and a database: HTML could be constructed on the fly depending on which client was asking for the page and the current state of a dataset. With a web application, a client and a server could have an ongoing relationship.

Web applications in the late 90s were impressive, but they were limited by HTTP. The protocol was designed for fetching documents, not managing program input and output, so web developers were struggling to mimic the complex interactions that graphical desktop applications had been enjoying for many years. Unless a webpage were refreshed, changes in the state of the server code could not be reflected in the interface. Complex JavaScript programs could be run on a page, but communication between the backend application and the JavaScript code was impossible. Backend data could be passed to a JavaScript program when it began executing, but new information could not

[57] "The Original HTTP as Defined in 1991." *W3C*. W3C, n.d. Web. 27 Nov. 2013. <http://www.w3.org/Protocols/HTTP/AsImplemented.html>.

be fetched and data could not easily flow the other direction. The web was essentially a one way street: once a page was pushed to the browser, it became isolated. The limitations of HTTP were a big problem for web developers in the late 90s. At the same time that complex desktop applications like Adobe Photoshop and Microsoft Excel were becoming the standard tools of entire industries, no one could write a decent email client for the web.

Ajax revolution.

In response to this problem, a software engineer at Microsoft named Alex Hoppman came up with a solution. At Microsoft in 1998, the Outlook team needed a good way to implement their email client for the web. The first version of Outlook Web Access (OWA) had been a mess, so they were determined to do it right. According to Hoppman's recount of OWA development on his blog, "There were two implementations that got started, one based on serving up straight web pages as efficiently as possible with straight HTML, and another one that started playing with the cool user interface you could build with [dynamic HTML and JavaScript]."[58] Unfortunately, the team doing the dynamic pages was having trouble doing server communication. "They were basically doing hacky form-posts back to the server," says Hoppman. Out of necessity, Hoppman implemented a new solution: "That weekend I started up Visual Studio and whipped up the first version of what would become XMLHTTP."[59] Hoppman's XMLHTTP allowed JavaScript code to make HTTP "GET" and "POST" requests to a web server and perform

[58] Hopmann, Alex. "The Story of XMLHTTP." *Alex Hopmann's Web Site*. N.p., n.d. Web. 24 Sept. 2013. <http://www.alexhopmann.com/xmlhttp.htm>.

[59] Ibid.

data transactions *without* refreshing the page. Although it did not seem revolutionary at the time, Hoppman's weekend project was a sea change for web application development.

The XMLHttpRequest object for JavaScript, providing an interface to this technology, was first introduced with Internet Explorer 5.[60] The library and the common best practices associated with it are now collectively called Ajax (Asynchronous JavaScript and XML). Aaron Swartz, the late web programmer and Internet activist, wrote a short history of the technology. According to Swartz, Ajax was not immediately popular. "Microsoft added a little-known function call named XMLHttpRequest to IE5. Mozilla quickly followed suit and, while nobody I know used it, the function stayed there, just waiting to be taken advantage of."[61] Although no one was using it at first, Ajax quietly changed all the basic assumptions that developers had been making about web applications. As Swartz said, "XMLHttpRequest allowed the JavaScript inside web pages to do something they could never really do before: get more data."[62]

The first company to use Ajax seriously for their core products was Google. In 2004, Google released a beta version of Gmail, their popular JavaScript email client. Gmail's easy to use interface that displayed emails as they arrived felt like a desktop application, with infinite cloud storage. The web development community took notice: Gmail proved that Ajax was technically sound and practical for real-world applications. Although it took a few years to take hold, by 2005, Ajax was a standard element of web development toolkits.

---

[60] Ibid.

[61] Swartz, Aaron. "A Brief History of Ajax." *Raw Thought*. N.p., 22 Dec. 2005. Web. 27 Nov. 2013. <http://www.aaronsw.com/weblog/ajaxhistory>.

[62] Ibid.

Supercharged JavaScript.

As web users continued to demand more complex experiences on the web, there was increased interest in building good tools and frameworks for JavaScript. Far and away, the most popular web framework for JavaScript was JQuery. According to a 2013 survey by World Wide Web Technology Surveys, JQuery is used in 56% of all websites (including 92% of websites that use *any* JavaScript framework).[63] JQuery, introduced in 2005, adds intuitive web-specific idioms on top of JavaScript, including built-in Ajax, CSS and JSON support. For example, the following Ajax code, which would be far more verbose in "vanilla" JavaScript, specifies that when a button is clicked, a new snippet of html will be loaded into part of the page:

```
$("#button").click(function(){
  $.get("html_snippet.html", function(data) {
    $("#mydiv").html(data);
  });
});
```

The rising popularity of Ajax and expressive frameworks like JQuery meant that JavaScript performance suddenly became important. When JavaScript use was limited to a few lines to tie together web components, the speed of that code's execution was not important. But by the mid-2000s, developers were writing software in JavaScript that was starting to hit the performance limits of browser interpreters. Google was leading the way with web application complexity and was therefore facing performance bottlenecks years ahead of other developers. In order to facilitate the complex interfaces of Google Maps,

[63] "Usage of JavaScript Libraries for Websites." *Web Technology Surveys*. W3Techs, n.d. Web. 27 Nov. 2013. <http://w3techs.com/technologies/overview/javascript_library/all>.

Gmail and its other web services, Google needed some serious JavaScript horsepower. And by 2006, it was clear to Google leadership that that kind of horsepower was not being supported in the browsers.[64]

The development of the Chrome browser, which began in 2006, was a strategic move on Google's part to force other browser makers to innovate. By building a better browser, Google leaders hoped they could push Apple, Microsoft and Mozilla to improve the web browsing experience and ultimately the performance of Google's products. JavaScript speed was the most important metric that Google targeted; Chrome's V8 JavaScript engine, named for the car engine, was designed to be extremely fast. According to Lars Bak, the lead of the V8 project, Google built the engine to "raise the performance bar of JavaScript out there, in the marketplace."[65]

As Google had hoped, soon after Chrome was launched in 2008, other browser makers were forced to make changes. The V8 engine, with just-in-time (JIT) JavaScript compilation and advanced garbage collection techniques, was much faster than anything else on the market. The other browser makers caught up quickly however. In 2009 Mozilla released Firefox 3.5 with TraceMonkey, a JIT JavaScript interpreter that was 40 times faster than its predecessor (SpiderMonkey).[66] Around the same time Apple introduced SquirrelFish Extreme to Safari, which was similar in performance metrics to

---

[64] Angwin, Julia. "Sun Valley: Schmidt Didn't Want to Build Chrome Initially, He Says." *The Wall Street Journal*. N.p., 9 July 2009. Web. 27 Nov. 2013. <http://blogs.wsj.com/digits/2009/07/09/sun-valley-schmidt-didnt-want-to-build-chrome-initially-he-says/>.

[65] "V8: An Open Source JavaScript Engine." *YouTube*. YouTube, 15 Sept. 2008. Web. 27 Nov. 2013. <http://www.youtube.com/watch?v=hWhMKalEicY>.

[66] Paul, Ryan. "Firefox to Get Massive JavaScript Performance Boost." *Ars Technica*. N.p., 22 Aug. 2008. Web. 27 Nov. 2013. <http://arstechnica.com/information-technology/2008/08/firefox-to-get-massive-javascript-performance-boost/>.

both TraceMonkey and V8.[67] Finally, not be outdone, Microsoft introduced the Chakra engine in its Internet Explorer 9 release.[68] The result was that JavaScript performance experienced an order of magnitude speedup over just a few years. As Brendan Eich, CTO of Mozilla, wrote on his blog in 2008, "Once we had launched TraceMonkey, and Apple had launched SquirrelFish Extreme, the world had multiple proofs along with the V8 release that [JavaScript] was no longer consigned to be 'slow' or 'a toy'"[69]

JavaScript on the server.

In 2006, a math student named Ryan Dahl, disenchanted with the world of academic mathematics, quit his PhD program and moved to South America.[70] In order to support himself, Dahl picked up programming and wrote web applications in PHP. At the time, the Ruby on Rails framework, which allowed developers to quickly write web servers in Ruby, was very new. Dahl became interested in Rails, but his interest quickly turned to annoyance. Rails was slow. In a 2011 interview, Dahl recounts thinking about the framework a lot during this time: "Rails had this problem. For every request coming into the server, it did a big lock. A request comes in, and until a response it made for this request, we're not going to do anything else... [Ruby on Rails] had zero concurrency."[71]

Out of frustration with Rails, Dahl began work on a project to write a non-

---

[67] Stachowiak, Maciej. "Introducing SquirrelFish Extreme." *Surfing Safari*. N.p., 18 Sept. 2008. Web. 28 Nov. 2013. <https://www.webkit.org/blog/214/>.
[68] "The New JavaScript Engine in Internet Explorer 9." *IEBlog*. N.p., 18 Mar. 2010. Web. 28 Nov. 2013. <http://blogs.msdn.com/b/ie/archive/2010/03/18/the-new-javascript-engine-in-internet-explorer-9.aspx>.
[69] Eich, Brendan. "New JavaScript Engine Module Owner." N.p., 21 June 2011. Web. 27 Nov. 2013. <https://brendaneich.com/2011/06/new-javascript-engine-module-owner/>.
[70] *History of Node.js*. Prod. Ryan Dahl. *YouTube*. YouTube, 05 Oct. 2011. Web. 24 Sept. 2013. <http://www.youtube.com/watch?v=SAc0vQCC6UQ>.
[71] Ibid.

blocking web server: one that could handle multiple incoming requests on a single thread. He quickly abandoned Ruby. “The problem with Rails, why it was so slow, wasn’t that they were doing something stupid in Ruby code, it was Ruby itself.”[72] Ruby, like Python and other interpreted languages, has a global interpreter lock, meaning it is not possible to execute multiple tasks simultaneously. In order to solve this problem, Dahl turned to other languages, including C and Haskell. Then, he says, it struck him. “Around January of 2009, I had this moment. Holy shit. JavaScript.”[73]

There were a few reasons that Dahl thought that JavaScript was the perfect candidate for writing what became NodeJS. First of all, JavaScript had the feature set he needed—anonymous functions and closures in particular. Secondly, JavaScript’s popularity was on the rise and people were interested in new projects in the language. Communities for JavaScript development were springing up all the time and the browser makers were doubling down on the language’s performance. Dahl notes that “V8 was released in December 2008. It was clear at that point that these big companies, Google, Apple, Microsoft and Mozilla, were going to be having an arms-race for JavaScript.”[74]

But the critical reason for choosing JavaScript, in Dahl’s mind, was that no one had really tried to put JavaScript on a server before. “No one had a preconceived notion of what it meant to be a server-side JavaScript program.”[75] There is no concept of input and output in JavaScript, let alone the concepts of interfacing with the file system or dealing with HTTP requests. No JavaScript libraries existed for performing any of these tasks, which Dahl thought was great. “If there had been [libraries for interfacing with the

[72] Ibid.
[73] Ibid.
[74] Ibid.
[75] Ibid.

operating system], they would have been implemented wrong... In a non-blocking system, as soon as you introduce a single blocking component, the whole system is screwed."[76] JavaScript allowed Dahl to start from scratch. He wrote all the building blocks of his JavaScript implementation so that they were non-blocking and could be run on an event-driven system.

Although Dahl's original intent with Node was to free web frameworks from global thread blocks, and he succeeded in that goal, the project has had a much larger impact. Node was the first serious implementation of JavaScript outside of a browser or a sandboxed application. This meant that the choices Dahl made when implementing it have impacted the language in a big way. For instance, Node adds a module system to JavaScript. This means that programmers who are writing code completely outside the context of the web, for instance scripts for doing mathematics calculations, can rely on modules written by other people:

```
var circle = require('./circle.js') // a module import
var radius = 4;
console.log(circle.area(radius));
```

Although Node was written for web server development, its standard libraries open up the entire operating system for JavaScript programs. The project has carved out a vast new territory for JavaScript developers, and it gives the language the tools to operate outside the browser. Dahl's work does not just free the dynamic web server from a global interpreter lock; it *frees JavaScript from the browser.* The effect on the language has been immeasurable.

---

[76] Ibid.

NextFive: Pure JavaScript Development in 2013

> *AtWood's Law: Any application that can be written in JavaScript, will eventually be written in JavaScript.* – Jeff AtWood, Coding Horror

Modern JavaScript is expressive and powerful. With JQuery and other browser-side libraries, developers can quickly write complex Ajax applications. With NodeJS, developers have a robust JavaScript implementation that runs on the server. The advances of the last decade challenge the idea that JavaScript is still a toy language. But has JavaScript graduated to the level of a "real" application programming language? Should a serious web developer in 2013 consider JavaScript as a true alternative to Ruby, Python, Scheme or even Java?

In order to test these questions, and push the limits of modern JavaScript, I joined a team of software developers to write a web application with only JavaScript. The Exavault team, based on Oakland, California builds web applications to help businesses share large files. For three months, I worked for one day a week with the five person development team at Exavault to build NextFive, a collaboration tool written entirely in JavaScript. By the end of the three months, we launched a beta version of the application good enough to be used internally. NextFive development at Exavault is ongoing.

Project requirements.

NextFive is a tool designed to solve a real problem that Exavault's software development team faces on a daily basis. Exavault's employees work remotely. Although the company is based in Oakland, only two employees regularly work from that location.

Exavault's way of doing business is increasingly common in the modern world, especially in the technology sector. Advances in code version control software and online meeting tools have allowed teams to work together from separate locations. But an online-only office environment sometimes causes inefficiencies. How can a team member keep track of what everyone else is working on?

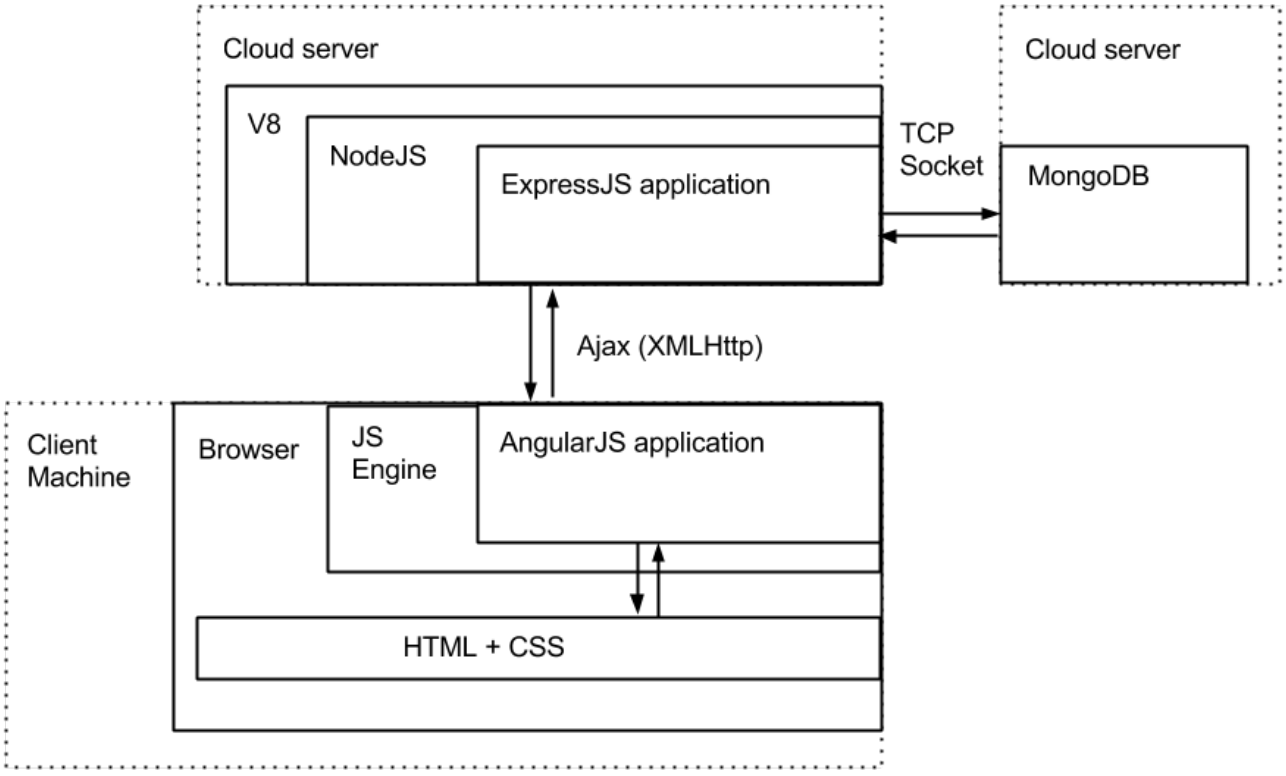
Remote teams have typically tried to solve this problem with email reports. At the beginning of each week, every employee writes a short report detailing what she plans to accomplish each day that week and shares it with the rest of the team. Making each individual contributor's workload easily visible to the rest of the team increases the chances that inconsistencies will be caught early. NextFive is a tool designed to make this reporting process smoother. Instead of writing email reports, users of NextFive can write reports in a web interface, compare reports online and keep track of reports over a longer period. In order to accomplish its goals, NextFive had to support secure login for employees, email integration, and interfaces for writing, reading and comparing reports. In September, 2013 we set out to build a beta version of NextFive in pure JavaScript.

Lean and MEAN.

To power our application we chose on a series of JavaScript technologies that together have been called the "MEAN" stack.[77] MEAN stands for MongoDB, ExpressJS, AngularJS and NodeJS. MongoDB is a database that allows for handling data in JavaScript Object Notation (JSON), ExpressJS is a web framework for NodeJS that

[77] "The MEAN Stack: MongoDB, ExpressJS, AngularJS and Node.js." *The MongoDB Blog*. N.p., 30 Apr. 2013. Web. 27 Nov. 2013. <http://blog.mongodb.org/post/49262866911/the-mean-stack-mongodb-expressjs-angularjs-and>.

handles web server boilerplate and AngularJS is a browser-side JavaScript framework designed for programming web interfaces using the Model-View-Controller (MVC) pattern. We chose the MEAN stack because the components work well together and the stack has been used by others in the JavaScript community. Although the components are very new—the first stable version of AngularJS was released in 2012—the MEAN stack has become a popular choice for writing pure JavaScript applications. The following is a rough sketch of the MEAN architecture:



The Angular application, which runs in the browser on a client machine, manages all user input and all HTML views, including rendering new views in response to user input. When more data is required, be it a snippet of HTML or a report object, Angular is responsible for making an Ajax request to the Express application. In turn, Express, which is running in NodeJS on a cloud server, is responsible for responding to Angular's

requests by creating, updating or fetching data from the Mongo database over a wire protocol. Although Express is responsible for handling user authentication and serving the Angular application to the browser, once the application has been initialized Express acts mostly as an interface for Angular to interact with data through HTTP requests, rather than as the main center of application logic.

In addition to the core technology stack, we relied on a number of open-source modules to help us abstract away critical application components. For example, we used PassportJS for safe user authentication, MongooseJS to provide data schemas on top of Mongo boilerplate and ExpressMailer to allow us to send emails from JavaScript code. These modules, available for free download, are becoming standards for NodeJS development. We managed these add-ons and their dependencies with Node's built in package manager, NPM. Lastly, we used a JavaScript build tool called GruntJS to help us automate development processes, such as checking for syntax errors and starting the server with the right configurations.

Challenges.

While it was fun to use a cutting edge set of technologies, the flip side of new tools is that issues and bugs have yet to be worked out. We found that this was especially true with Angular, the most important and newest component in our stack. Unfortunately, the official documentation for Angular is lacking; for a heavyweight framework like Angular, this means a very steep learning curve. Although we were prepared for this eventuality when we started, the lack of documentation on full-stack JavaScript tools did make the development process slower. The tools were powerful, but coding speed was

variable.

Another major challenge we faced was adapting to a NodeJS way of writing server code. For example, Node is designed so that all program input and output is done asynchronously, meaning that all uses of input must be handled in callback functions. This demands an entirely different way of programming and reasoning about code. The following Python code will serve as a good illustration of the differences between Node and other server languages:

```
import sqlite3

db_cursor = sqlite3.connect('test.db').cursor()
db_cursor.execute('SELECT SQLITE_VERSION()')
version = db_cursor.fetchone()
## The database query is complete by this point
print "SQLite version: %s" % version
```

This snippet of code connects to a database and performs a simple query. It is correct and easy to read, but it has some hidden costs. A computer can perform a computation on register value in one CPU cycle, but some amount of time must be spent retrieving data if it needs to be put into a register first. For instance, it costs time equal to about 3 CPU cycles for a program to go to the L1 cache to fetch data.[78] Similarity, it costs 14 cycles to go to the L2 cache, 250 to go to RAM, 41 million to go to disk and 250 million (or more) to fetch data over the network.[79] When the Python program is executing the database query (on line 5) the program is completely halted. The print statement on the final line is not executed until the database query returns and the ‘version’ variable has been assigned.

[78] Takada, Mikito. "Understanding the Node.js Event Loop." *Mixus Tech Blog*. N.p., 1 Feb. 2011. Web. 27 Nov. 2013. <http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/>.

[79] Ibid.

This is very convenient for the programmer: she can treat assigning a variable to the result of a database query the same way she would treat any other type of assignment. But if the database query takes a billion CPU cycles to complete, that's a billion cycles that the program must idle in order to make that abstraction possible. This is an enormous waste of computation resources.

This is precisely the problem that Ryan Dahl had with Ruby when he decided to write NodeJS in 2009. As a response to this kind of programming, all of NodeJS's external calls are done asynchronously. This breaks the assignment abstraction: assigning a variable to a JavaScript expression is not the same as assigning a variable to the results of a database query in Node. For instance, the following Python-style NodeJS code does not do what we might expect:

```
var user = User.findOne({name: "Brendan Eich"});
console.log(user);
```

NodeJS does not stop on the first line to perform the database query. Instead it sends a request for the query to complete and then moves on, executing the log statement. This means that the log statement on line 2 prints nonsense. To query a database in Node, we have to push a "callback" function onto Node's event loop that will be called when the query returns. We can only rely on the return value of the query being defined *within* the callback. In this example, logResult will be called when User.findOne returns, but after other code is executed:

```
function logResult(err, db_result){
```

```
  console.log(db_result)
}

User.findOne({name: "Brendan Eich"}, logResult)
console.log("This is printed before logResult is called.");
```

This kind of programming is everywhere in NodeJS and it can lead to some nasty looking code. In the NextFive codebase, the hundreds of anonymous functions defined as callbacks (sometimes many levels deep) handle more responsibility and logic than the named functions do. Asynchronous programming is fundamentally different from writing Python, or PHP. It took the NextFive team, myself included, some time to get used to it.

Advantages.

Using one language for the entire application was a powerful simplification. When implementing a new feature for a web application, it is often necessary to write code at multiple levels of the stack. For instance, if we wanted to add the ability for a user to highlight a NextFive report with a star, this would require adding a ‘star’ field to the data in the database, writing logic to handle that extra field on the server, and handling when that field gets filled on the front-end. This requires a significant amount of context switching for the programmer if the language is different at each one of these levels. In a polyglot stack, what tends to happen is that majority of the logic gets pushed to one side—typically the server—while the other side is “spoon fed” already processed information. This allows for the least amount of context switching for the programmer who can mostly stick to one side of the stack. But in a pure JavaScript application, the language is the same throughout. So there are fewer advantages to pushing application

logic one way or another. In fact, during NextFive development we found that we were blurring the lines between browser and server code all the time. We wrote functions that could be run on either side of the divide to the same effect, so it did not matter where we put them. This meant that for some features of NextFive, the backend handles almost all the logic and spoon-feeds the front-end just the information it needs to render the interface. For other features, the backend sends raw data down the line and Angular handles the logic and computation. Being able to push logic to both sides of the stack, without straining the brains of the programmers, sped up development time significantly.

Another significant advantage of the same language everywhere is that the data has the same representation throughout the application. In a MEAN stack, Express queries Mongo with a JSON query and is returned a JSON object. Express then handles that object and passes it as JSON down to Angular, which can understand the data with no extra interface code, and finally the data object is rendered in HTML with dot notation similar to JSON. The same data format at every level means we can think of the data in the same way throughout the codebase. This simplification makes building a mental model of the data much easier. In the NextFive codebase, a report object is a report object is a report object, no matter where you handle it.

Conclusions and comparisons.

NextFive shows that pure JavaScript development, particularly with the MEAN stack, is comparable to development in dynamically typed languages like Python, Ruby or PHP. In some ways, pure JavaScript development is superior. For instance, JavaScript runs in the browser; no other language can provide a programmer the clean abstraction of

one language on the backend and front-end. In other ways, pure JavaScript is not as good. The popular dynamic web languages all boast battle-tested web frameworks supported by large communities. Pure JavaScript developers are largely on their own. However, after writing a significant application in pure JavaScript, the Exavault team and I did not feel that another language would have been a better choice. In 2013, the prudent web programmer should consider pure JavaScript a reasonable alternative to a dynamic web stack.

Unfortunately, NextFive does not test whether a pure JavaScript application could scale to millions of users, hundreds of thousands of lines of code or hundreds of programmers. That kind of development is a magnitude larger than what we were able to study given the constraints of the project. This is particularly unfortunate because in order to test whether or not JavaScript could replace Java or C++, JavaScript must be tested at that level: the advantages of compiled languages become clear at scale. However, even without testing, it is difficult to imagine a case in which backend JavaScript (rather than C++) would be the smart choice for an application the size of Gmail or Bing Search. JavaScript is dynamically typed and cannot offer software engineering teams the tools they need to succeed on that level. This does not take away from JavaScript however. The fact that the language might even be considered as an alternative to C++ shows that the modern language is far superior to the language that Brendan Eich wrote in ten days. JavaScript is no longer a toy language.

The Future of JavaScript

> *Like music and food, a programming language can be a product of its time. The deep problem in language design is not technological, it is psychological.* – Douglas Crockford

It is an exciting time to be a JavaScript developer and many are optimistic about the future of the language, for good reason. On the other side of the coin, however, JavaScript is far from perfect. Sure, critics contend, the language has matured considerably since its birth, but it had a lot of maturing to do. Some have suggested that JavaScript is only a reasonable programming language in comparison to its past self. Most pessimistically, some critics have accused pro-JavaScript developers of having "Stockholm Syndrome" after being forced to write the language for so many years.[80]

Debates surrounding JavaScript are far from settled; the most potent of these debates is how to move forward. What will web development be like in 2023? What *should* it be like? These questions could not be more important for millions of businesses built around the web. Google, Microsoft and Mozilla, the popular browser makers and most influential forces in the JavaScript community, each answer these questions differently. These major political forces in the JavaScript community have articulated competing visions for the future of the web and JavaScript. The next chapter in the history of the web will not be written by one company or group, but its trajectory will likely be bent by the trajectory of JavaScript. And the trajectory of JavaScript will be influenced a great deal by the major browser makers.

---

[80] "Node.JS: The Good Parts? A Skeptic's View." *YouTube*. YouTube, 21 June 2013. Web. 27 Nov. 2013. <http://www.youtube.com/watch?v=CN0jTnSROsk>.

JavaScript at scale.

Developers are demanding more and more from JavaScript. The language is still used for short scripts, but some JavaScript applications are becoming big enough that one person cannot reasonably understand them. Unfortunately, despite its successes, JavaScript remains problematic. Object-oriented programming in JavaScript is still done without classes, functional programming in JavaScript is still done without pattern matching and there is still no integer type in JavaScript. In modern JavaScript, the equality and plus operators still do confusing auto type-conversion. For simple scripts, that kind of behavior might not be a problem; in fact, it might be welcome for a developer who wants to work very quickly and avoid the hassles of explicit type casting and run-time errors. But when programming in the large, JavaScript's weaknesses compound themselves.

The problems that arise at scale are not new in software development. Large teams of engineers have written enormous systems in Java, C++ and other compiled languages. Unlike dynamic languages however, in these languages programmers can rely on tools to analyze their code. Static analysis tools allow programmers to make guarantees about what could happen when a C++ program is run, without running it. This kind of tooling is not really possible in a dynamic language. The responsibility for type checking in Python, Ruby and JavaScript is pushed from the language onto the programmer. Attempting to do some level of static analysis on dynamic languages is an open area of research in Computer Science, but results have been limited.[81] If the web is going to keep growing, and allow support for increasingly complex applications that can

[81] Wiedermann, Ben, Vineeth Kashyap, John Sarracino, John Wagner, and Ben Hardekopf. "Type Refinement for Static Analysis of JavaScript." (2013): n. pag. Print.

compete with desktop applications, it will have to support programming in the large. Although Ecma has continued to address the problems with JavaScript incrementally, their latest specification, ECMAScript 6, which adds block scoping and constant declarations among other improvements, will do little to solve the problems of writing JavaScript at scale.[82]

Incremental improvement.

Microsoft's vision for the future of JavaScript, an open-source project called TypeScript, addresses the scale limits of dynamic JavaScript head-on. TypeScript is a strict superset of JavaScript that compiles into JavaScript.[83] This means that JavaScript programs are automatically TypeScript programs and that TypeScript programs can be integrated seamlessly into the existing JavaScript ecosystem. According to TypeScript's website, "With TypeScript, you can use existing JavaScript code, incorporate popular JavaScript libraries, and be called from other JavaScript code."[84] Microsoft is making an effort to make TypeScript as attractive as possible for those who write JavaScript, including Node developers. TypeScript's marketers boast that "The output of the TypeScript compiler is idiomatic JavaScript."[85]

Although Microsoft wants TypeScript to read and write like JavaScript, their

---

[82] "Draft Specification for ES.next." *ECMAScript Wiki*. N.p., n.d. Web. 27 Nov. 2013. <http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts>.

[83] "TypeScript." *Welcome to TypeScript*. Microsoft, n.d. Web. 27 Nov. 2013. <http://www.typescriptlang.org/>.

[84] Ibid.

[85] Bright, Peter. "Microsoft TypeScript: The JavaScript We Need, or a Solution Looking for a Problem?" *ArsTechnica*. N.p., 2 Oct. 2012. Web. 28 Nov. 2013. <http://arstechnica.com/information-technology/2012/10/microsoft-typescript-the-javascript-we-need-or-a-solution-looking-for-a-problem/>.

ultimate goal is to address the problems of web programming at scale. The project lead, Anders Hejlsberg, who is also the lead architect of C#, articulated Microsoft's goals with TypeScript in a talk at Microsoft's Build conference in 2013. "People are finding it very hard to write large applications in JavaScript... we hear this both externally and internally."[86] Hejlsberg says that JavaScript was never designed to write large scale applications. "It was intended for hundred line apps, and now with regularity we are writing hundred-thousand line apps... JavaScript does not have any large scale application concepts in it."[87]

TypeScript is designed for programming in the large. In contrast to other attempts to improving JavaScript, which often approach the language as a functional language, Microsoft is taking the aspects of JavaScript that are similar to Java and C# and emphasizing them. The company is attempting to add concepts from those languages to JavaScript, meaning that those who are familiar with both JavaScript *and* Java will find TypeScript familiar:

```
// A generic TypeScript class
class Greeter<T> {
  greeting: T;
  constructor(message: T) {
    this.greeting = message;
  }
  greet() {
    return this.greeting;
  }
}

var greeter = new Greeter('Hello world');
```

[86] Hejlsberg, Anders, and Steve Lucco. "TypeScript: Application-Scale JavaScript." *MDSN*. Microsoft, 26 June 2013. Web. 28 Nov. 2013. <http://channel9.msdn.com/Events/Build/2013/3-314>.

[87] Ibid.

TypeScript is (optionally) statically typed and supports classes, generics, modules and interfaces among other modern language concepts. The inclusion of static types means that TypeScript can be reasoned about by a machine. Unlike JavaScript, TypeScript can be tooled by intelligent environments that allow for automated refactoring and other benefits of code comprehension. Hejlsberg thinks intelligent tools is the critical difference for programmers who want to write large systems: “If you think about what it is that has powered the revolution we have seen in intelligent development environments over the last ten years, most of the power comes from static typing.”[88] With TypeScript, Microsoft hopes, code comprehension can bring to JavaScript what it has already brought to other languages: power and control when programming very large applications.

Abstraction and avoidance.

At the core of the TypeScript project is the idea that JavaScript can be abstracted away from the programmer: if JavaScript is the compile target of a nicer language, then people can write that nicer language and still run their programs in the browser. This is not a new idea. Programmers have tried, with mixed success, to transcompile almost every modern programming language into JavaScript. CoffeeScript, a language designed to be transcompiled into JavaScript, has arguably had the most success. Introduced in 2010, by 2012 CoffeeScript was the 13th most popular language on the popular code hosting site Github.[89]

[88] Hejlsberg, Anders, and Steve Lucco.
[89] Brockmeier, Joe. "RedMonk Programming Language Rankings: CoffeeScript and Java Make Gains."

CoffeeScript looks like a far more functional and far less verbose version of JavaScript. Unlike TypeScript, which approaches JavaScript as if it was C#, CoffeeScript approaches JavaScript as if it was Ruby. CoffeeScript attempts to strip out the JavaScript's verbosity and present a clean, functional language. According to the language's website, "Underneath that awkward Java-esque patina, JavaScript has always had a gorgeous heart. CoffeeScript is an attempt to expose the good parts of JavaScript in a simple way."[90]

```
// CoffeeScript
square = (x) -> x * x
alert "A exists" if A?


// transcompiled JavaScript:
(function() {
  var square;

  square = function(x) {
    return x * x;
  };

  if (typeof A !== "undefined" && A !== null) {
    alert("A exists");
  }

}).call(this);
```

Brendan Eich, the CTO of Mozilla, has said that CoffeeScript influences how he thinks about JavaScript.[91] And Eich is (to some extent) a supporter of the core philosophy

---

ReadWrite. N.p., 14 Feb. 2012. Web. 28 Nov. 2013. <http://readwrite.com/2012/02/14/redmonk-programming-language-r>.

[90] "CoffeeScript." *CoffeeScript*. N.p., n.d. Web. 28 Nov. 2013. <http://coffeescript.org/>.

[91] Eich, Brendan. "My JSConf.US Presentation." N.p., 4 May 2011. Web. 28 Nov. 2013. <https://brendaneich.com/tag/javascript-ecmascript-harmony-coffeescript/>.

behind both TypeScript and CoffeeScript: one of his goals for ECMAScript 6 is for the language to be "a better language for writing code generators."[92] But unfortunately, CoffeeScript does not solve the problems of writing JavaScript at scale. CoffeeScript is designed to make JavaScript more human readable, but it does not address the machine readability of the language. Mozilla, under Eich's leadership, has a bolder vision for the future of JavaScript that does address the concerns of large-scale web programming. In the pursuit of performance and scale, why not *completely* abstract JavaScript away from the programmer?

Emscripten, a Mozilla research project, compiles C and C++ code to JavaScript. Unlike TypeScript's compiler however, the output of Emscripten, called ASM.js, is not intended to be read by humans. Instead, ASM.js is a strict subset of JavaScript intended to act as an assembly language for the browser. According to Mozilla, "Validated asm.js code is limited to an extremely restricted subset of JavaScript that provides only strictly-typed integers, floats, arithmetic, function calls, and heap accesses."[93] Instead of using objects like traditional JavaScript, ASM.js programs manipulate a large global array that represents memory, where object data can be stored. Instead of allowing any variable type in the program, ASM.js only operates on integer-like JavaScript numbers (it can force this behavior by doing a bitwise "or" operation with zero on every number in the program).

---

[92] Eich, Brendan. *The Harmony Goals*. Digital image. N.p., n.d. Web. 27 Nov. 2013. <https://brendaneich.com/brendaneich_content/uploads/ES.003.png>.

[93] "Asm.js - Frequently Asked Questions." *Asm.js*. Mozilla, n.d. Web. 28 Nov. 2013. <http://asmjs.org/faq.html>.

```
// calculate the length of C string in ASM.js
function strlen(ptr) {
  ptr = ptr|0;
  var curr = 0;
  curr = ptr;
  while (MEM8[curr]|0 != 0) {
    curr = (curr + 1)|0;
  }
  return (curr - ptr)|0;
}
```

The result of the extreme restrictions placed on ASM.js is that the language subset operates at a much lower level than traditional JavaScript programs. It also means that ASM.js programs can be analyzed statically. ASM.js is just JavaScript, so it can be interpreted by any modern JavaScript interpreter. But it does not *have to be*. A smarter JavaScript engine, which Mozilla is working on, could recognize ASM.js and perform ahead-of-time compilation to machine code, skipping the interpretation process all together.[94]

The consequences of Emscripten and ASM.js are three fold. First, code in the browser can be extremely fast. According to Mozilla, their early benchmarks of ASM.js are within a factor of two over native compilation of C.[95] That would mean a two to ten times speedup over regular JavaScript programs running on the most advanced JavaScript interpreters today. Such a speedup would allow for new types of programs to run on the web. For instance, immersive video games with complex 3D graphics, which for performance reasons are often written in C++, could be run on the web. In fact, Mozilla has demoed this capability to the public.[96] Secondly, developers could write large-scale

[94] Ibid.
[95] Ibid.
[96] "Unreal Engine 3 in Firefox with Asm.js." *YouTube*. YouTube, 02 May 2013. Web. 28 Nov. 2013.

web applications in a language designed for writing big systems. The problems that arise when writing large applications in C++ have been solved over the last two decades. Tools, patterns and best practices already exist for that kind of programming. Lastly, the web could become language neutral. Emscripten compiles C and C++ to JavaScript by compiling it first into LLVM using a C compiler, and then compiling LLVM into JavaScript. LLVM (Low Level Virtual Machine) is a low level intermediate language (similar to Java bytecode) that is targeted by a number of language compilers. Therefore, in theory, future web developers could write Java, Scala, Python, Ruby, C#, Ada, Objective-C or Haskell for the web. In the future, if all browsers supported ahead-of-time ASM.js compilation, we would not even need the ASM.js specification. We could replace ASM.js with something nicer that provided the same functionality. But for the time being, ASM.js is *just JavaScript*, so it can be a solution today.

Replacing JavaScript.

The implicit assumption behind TypeScript and Emscripten is that JavaScript is not going away anytime soon. Even if we want to write other, more scalable languages for the web, eventually JavaScript code has to actually be run in the browser. Google rejects this assumption. Google's wants the world to stop trying to engineer around JavaScript, and instead simply accept that the language is broken and can be replaced.

Google's vision for the future of JavaScript was made public when an internal memo was leaked in 2010. According to the memo, which represents the consensus of many JavaScript leaders at Google, including representatives from the ECMAScript

---

<http://www.youtube.com/watch?v=BV32Cs_CMqo>.

standards body and the V8 project, JavaScript has fundamental flaws that harm the web. "Javascript as it exists today will likely not be a viable solution long-term."[97] The document advocates for the company to pursue a two-pronged approach to the language. First, Google would continue to support the evolution of ECMAScript. "It is paramount that Google continue to maintain a leadership position on important open web standards such as [ECMAScript 6]."[98] The authors of the document argue that supporting ECMAScript publicly is a low-risk, low-reward option for Google. "The 'evolve Javascript' option is relatively low risk, but even in the best case it will take years and will be limited by fundamental problems in the language (like the existence of a single Number primitive)."[99] To augment this option, the authors argue that Google should also pursue a high-risk, high-reward approach in parallel: Google would develop a language to replace JavaScript.

Dart, released in November 2013, is the language that leaked memo referred to. According to the memo, and current documentation of Dart, the language is designed with three priorities in mind. First, performance. Google's V8, Mozilla's TraceMonkey, Microsoft's Chakra and other JavaScript engines have made JavaScript about as fast as it can be as a dynamic language. With a fresh start, Google could optimize for speed from the beginning and avoid the problems that make optimizing JavaScript a full-time job for browser makers. Second, Dart is designed to focus on usability. The authors of the 2010 argue that keeping Dart easy to learn and fast to write is key to its widespread adoption. "[Dart] is designed to keep the dynamic, easy-to-get-started, no-compile nature of

---

[97] Miller, Mark. ""Future of Javascript" Doc from Our Internal "Javascript Summit"" *GitHub Gists*. Google, n.d. Web. 24 Sept. 2013. <https://gist.github.com/paulmillr/1208618>.
[98] Ibid.
[99] Ibid.

Javascript that has made the web platform the clear winner for hobbyist developers."[100] JavaScript can be written by amateurs without complex tools or training. Google believes this aspect of the web is important to maintain. And lastly, Google wants to solve the problems of web programming at scale. "[Dart] is designed to be more easily tooled (e.g. with optional types) for large-scale projects that require code-comprehension features such as refactoring and finding callsites."[101]

```
// A Dart class
class Fibonacci {
  fib(n) => n < 2 ? n : (fib(n-1) + fib(n-2));

  // Static types when we want them
  int static_fib(int n) {
    if (n < 2) {
      return n;
    } else {
      return static_fib(n-1) + static_fib(n-2);
    }
  }
}
```

Google views Dart as a fresh start for the web. The aim of the Dart team is to create a language designed from the ground up for scalable web application development, including on the server (similar to NodeJS). Although the engineering hacks that have gotten JavaScript to where it is today are impressive, Dart developers might argue, there is no reason why good engineers should be spending time on that, rather than building world class applications in a language designed for the task.

[100] Ibid.
[101] Ibid.

Winning the future.

Like Google, Microsoft and Mozilla support the ongoing work of Ecma. All three companies have members on the standards board and publicly endorse the group's processes and released standards. ECMAScript 6 will be the most significant change to the JavaScript since it was introduced. Unfortunately, it will not fix the problems of JavaScript development at scale and it is unclear whether web developers can wait. Like Google, Mozilla and Microsoft seem to be approaching the problem with two-pronged strategies. Beyond the continued evolution of JavaScript, Microsoft wants to improve JavaScript development with TypeScript, Mozilla wants to compile better languages into JavaScript with Emscripten and Google wants to replace JavaScript with Dart. Although each of these approaches involves serious technical issues, it is the political issues that define them. Reactions to the high-risk, high-reward solutions articulated by the major players in the JavaScript community have been mixed.

TypeScript works today. It integrates with the current JavaScript ecosystem but allows for programming large scale applications with idioms and structures that are familiar to Java and C# developers. Like CoffeeScript, TypeScript is a close cousin of JavaScript. Unlike CoffeeScript, TypeScript will open up a whole new world for JavaScript developers: intelligent environments. Unfortunately, TypeScript is still JavaScript. After compiling TypeScript, developers still have to run JavaScript in the browser, which is not nearly as fast as compiled languages. With TypeScript, could Photoshop be rewritten for the web? Immersive 3D games? No. At least not in the short term. TypeScript might bring web development ahead a few years, but what is Microsoft's vision for the web in 2023?

Mozilla's Emscripten and ASM.js would allow developers to write in scalable languages *and* execute code in the browser at near native speed. Mozilla's proposal means that the web could become just another compile target for programmers, or as they hope, the only compile target needed. Emscripten is an elegant solution because it allows for developers to write these types of applications today: ASM.js is just JavaScript, so any browsers can choose to compile it, but old browsers can still interpret it. However, Emscripten has its own share of problems. ASM.js is so stripped down that it cannot make calls to browser APIs or take advantage of a decade's worth of tools for JavaScript in the browser. We can compile million-line C++ video games for the browser with Emscripten, but could Google Docs be easily ported? Emscripten as it is today cannot replace JavaScript. Mozilla's solution is fantastic, but only for certain types of programs: when we want really fast, but completely isolated, computation in the browser.

Rather than deal with these problem, Google asks the web developer to imagine a world in which developers got the chance to erase some of the messy history of the web. In Google's world, we could write browser APIs designed for getting real work done. We could write a language for the web that treats the web as a real platform and tools for that platform that allow for professionals and amateurs alike to write expressive code. What if the web had modules, classes, package managers, IDEs, debuggers, static analyzers and operator overloading, Google asks. What if the web was fast, without the hassle? Google's vision is very appealing, and the Mountain View company is right: Dart is a superior language to JavaScript. But Google's world might be a dream. Dart breaks the modern web developer's toolkit. All of the tools built for JavaScript development over the past decade would have to be rewritten to work with Dart. Although Dart can be

compiled into JavaScript, in order to get its full power all other browsers would have to implement the language. Why would Microsoft, Apple or Mozilla ever implement Dart? It would only serve to make Google a stronger force in the web community; something that runs counter to the interests of other browser makers. Dart might be like C#. C# was Microsoft's attempt to write a better version of Java. The language is technically superior to Java in a number of ways, but it never achieved the community that Java had. Java developers rejected C#. Will JavaScript developers accept Dart?

It is important to remember that the web is not the only graphical gateway to the Internet anymore. Proprietary solutions like Apple's iOS platform allow for users to use the Internet without web technologies like HTML and JavaScript. Because the platform is entirely controlled by Apple, iOS is able to move quickly without being bogged down by the political debates of the JavaScript community. The web was the first interface for the public to use to the Internet, but will it be the last? Perhaps web development will not exist in 2023. And JavaScript will be a strange historical language that no one bothers to learn. The political leaders in the web community are fighting to make sure that is not a reality. The chaotic beauty of the web is that it is not controlled by anyone; the web is controlled by everyone. No one's website has to be approved by Apple.

As JavaScript leaders at Google remarked in the 2010 memo, "The web has succeeded historically to some extent in spite of the web platform, based primarily on the strength of its reach."[102] The web, which gave regular people the power to share information on a global scale, is one of the most culturally central technologies humankind has built. The power of the web was ground breaking in 1995. But in 2013, it

[102] Miller

is the status quo. With an influx of alternatives, there is no reason to assume that the future will be built on the open web. Instead, like every other platform, the web will be judged on its ability to deliver services and products that people want to use; and it will not be able to compete if HTML, HTTP, CSS and JavaScript hold it back. The future of the web is tied to the murky and political future of JavaScript. Because of its deep connection to the web, in spite of its humble beginnings in Brendan Eich's cubicle at Netscape, JavaScript is suddenly the most important language of the modern era: if JavaScript cannot support the future of the web, either the language or the platform will be left behind.

Bibliography

Angwin, Julia. "Sun Valley: Schmidt Didn’t Want to Build Chrome Initially, He Says." *The Wall Street Journal*. N.p., 9 July 2009. Web. 27 Nov. 2013. <http://blogs.wsj.com/digits/2009/07/09/sun-valley-schmidt-didnt-want-to-build-chrome-initially-he-says/>.

"AltJS - Web Coding beyond JavaScript." *Altjs.org*. N.p., n.d. Web. 24 Sept. 2013. <http://altjs.org/>.

Arthur, Charles. "Mobile Internet Devices Will Outnumber Humans This Year." *The Guardian*. N.p., 7 Feb. 2013. Web. 27 Nov. 2013. <http://www.theguardian.com/technology/2013/feb/07/mobile-internet-outnumber-people>.

"Asm.js - Frequently Asked Questions." *Asm.js*. Mozilla, n.d. Web. 28 Nov. 2013. <http://asmjs.org/faq.html>.

"Asm.js Specification." *Asm.js*. Mozilla, n.d. Web. 24 Sept. 2013. <http://asmjs.org/spec/latest/>.

Battelle, John. "Wired 13.08: The Birth of Google." *Wired.com*. Conde Nast Digital, Aug. 2008. Web. 27 Nov. 2013. <http://www.wired.com/wired/archive/13.08/battelle.html>.

"Blink." *Mozilla Developer Network*. Mozilla, n.d. Web. 27 Nov. 2013. <https://developer.mozilla.org/en-US/docs/Web/HTML/Element/blink>.

Byous, Jon. "JAVA TECHNOLOGY: THE EARLY YEARS." Sun Microsystems, 1998. Web. 27 Nov. 2013. <http://web.archive.org/web/20050420081440/http://java.sun.com/features/1998/05/birthday.html>.

A Brief History of Web Standards. Digital image. Vitamin T, 30 Oct. 2011. Web. 27 Nov. 2013. <http://visual.ly/brief-history-web-standards>.

Bright, Peter. "Microsoft TypeScript: The JavaScript We Need, or a Solution Looking for a Problem?" *ArsTechnica*. N.p., 2 Oct. 2012. Web. 28 Nov. 2013. <http://arstechnica.com/information-technology/2012/10/microsoft-typescript-the-javascript-we-need-or-a-solution-looking-for-a-problem/>.

Brockmeier, Joe. "RedMonk Programming Language Rankings: CoffeeScript and Java Make Gains." *ReadWrite*. N.p., 14 Feb. 2012. Web. 28 Nov. 2013. <http://readwrite.com/2012/02/14/redmonk-programming-language-r>.

Camp, Tony. "Ryan Dahl's History of Node.js." *Phx Tag Soup*. N.p., 5 Oct. 2011. Web. 27 Nov. 2013. <http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/>.

Clark, Andrew. "How Jerry's Guide to the World Wide Web Became Yahoo." *The Guardian*. N.p., 1 Feb. 2008. Web. 27 Nov. 2013. <http://www.theguardian.com/business/2008/feb/01/microsoft.technology>.

"CoffeeScript." *CoffeeScript*. N.p., n.d. Web. 28 Nov. 2013. <http://coffeescript.org/>.

Crockford, Douglas. "JavaScript: The World's Most Misunderstood Programming Language." Web log post. *Crockford.com*. N.p., n.d. Web. 24 Sept. 2013. <http://www.crockford.com/javascript/javascript.html>.
Crockford, Douglas. "JSON: The Fat-Free Alternative to XML." *Json.org*. N.p., n.d. Web. 24 Sept. 2013. <http://www.json.org/fatfree.html>.
Crockford, Doulgas. "The World's Most Misunderstood Programming Language Has Become the World's Most Popular Programming Language." Web log post. *Crockford.com*. N.p., Mar. 2008. Web. 24 Sept. 2013. <http://javascript.crockford.com/popular.html>.
"Draft Specification for ES.next." *ECMAScript Wiki*. N.p., n.d. Web. 27 Nov. 2013. <http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts>.
"DJS - Defensive JavaScript." *Defensivejs.com*. N.p., n.d. Web. 24 Sept. 2013. <http://www.defensivejs.com/>.
"ECMAScript Language Specification 3." Ecma International, Dec. 1999. Web. <http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf>.
"ECMAScript Language Specification 5.1." Ecma International, 2011. Web. 27 Nov. 2013. <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>.
Eich, Brendan. *The Harmony Goals*. Digital image. N.p., n.d. Web. 27 Nov. 2013. <https://brendaneich.com/brendaneich_content/uploads/ES.003.png>.
Eich, Brendan. "JavaScript: In Which 10 Days of May Did Brendan Eich Write JavaScript (Mocha) in 1995?" *Quora*. N.p., 16 Feb. 2013. Web. 27 Nov. 2013. <http://www.quora.com/JavaScript/In-which-10-days-of-May-did-Brendan-Eich-write-JavaScript-Mocha-in-1995>.
Eich, Brendan. "My JSConf.US Presentation." N.p., 4 May 2011. Web. 28 Nov. 2013. <https://brendaneich.com/tag/javascript-ecmascript-harmony-coffeescript/>.
Eich, Brendan. "New JavaScript Engine Module Owner." N.p., 21 June 2011. Web. 27 Nov. 2013. <https://brendaneich.com/2011/06/new-javascript-engine-module-owner/>.
Eich, Brendan. "Popularity." Web log post. N.p., 3 Apr. 2008. Web. 27 Nov. 2013. <https://brendaneich.com/2008/04/popularity/>.
Eich, Brendan. "The State of Javascript." *Brendaneich.github.io*. N.p., 2012. Web. 24 Sept. 2013. <http://brendaneich.github.io/Strange-Loop-2012/>.
Eich, Brendan. "Will There Be a Suggested File Suffix for Es4?" *Es4-discuss*. Mozilla, 3 Oct. 2006. Web. 27 Nov. 2013. <https://mail.mozilla.org/pipermail/es-discuss/2006-October/000133.html>.
"Evolving ECMAScript." *Blogs.msdn.com*. Microsoft, 22 Nov. 2011. Web. 24 Sept. 2013. <http://blogs.msdn.com/b/ie/archive/2011/11/22/evolving-ecmascript.aspx>.
"Fluent 2012: Brendan Eich, "JavaScript at 17"" *YouTube*. YouTube, 30 May 2012. Web. 27 Nov. 2013. <http://www.youtube.com/watch?v=Rj49rmc01Hs>.
Gruener, Wolfgang. "Google Dev: We Are Making Chrome Out of Kindness to Web." *Tom's Hardware*. N.p., 27 Dec. 2011. Web. 27 Nov. 2013. <http://www.tomshardware.com/news/google-chrome-web-browser-mozilla-firefox,14378.html>.

Hamilton, Naomi. "The A-Z of Programming Languages: JavaScript." *Computerworld*. N.p., 31 July 2008. Web. 27 Nov. 2013. <http://www.computerworld.com.au/article/255293/a-z_programming_languages_javascript/>.

Hanselman, Scott. "JavaScript Is Assembly Language for the Web Part 2: Madness or Just Insanity?" *Hanselman.com*. N.p., July 2011. Web. 24 Sept. 2013. <http://www.hanselman.com/blog/JavaScriptIsAssemblyLanguageForTheWebPart2MadnessOrJustInsanity.aspx>.

Hanselman, Scott. "JavaScript Is Web Assembly Language and Thats OK." *Hanselman.com*. N.p., n.d. Web. 24 Sept. 2013. <http://www.hanselman.com/blog/JavaScriptIsWebAssemblyLanguageAndThatsOK.aspx>.

Hejlsberg, Anders, and Steve Lucco. "TypeScript: Application-Scale JavaScript." *MDSN*. Microsoft, 26 June 2013. Web. 28 Nov. 2013. <http://channel9.msdn.com/Events/Build/2013/3-314>.

History of Node.js. Prod. Ryan Dahl. *YouTube*. YouTube, 05 Oct. 2011. Web. 24 Sept. 2013. <http://www.youtube.com/watch?v=SAc0vQCC6UQ>.

Hopmann, Alex. "The Story of XMLHTTP." *Alex Hopmann's Web Site*. N.p., n.d. Web. 24 Sept. 2013. <http://www.alexhopmann.com/xmlhttp.htm>.

"How Is JavaScript Different from Java?" *Java*. Oracle, n.d. Web. 27 Nov. 2013. <http://www.java.com/en/download/faq/java_javascript.xml>.

Hunt, Oliver. "WebKit Branch to Support Multiple VMs (e.g., Dart)." Email Mailing List. *Webkit-dev*. N.p., Dec. 2011. Web. 24 Sept. 2013. <https://lists.webkit.org/pipermail/webkit-dev/2011-December/018787.html>.

"Internet Growth Statistics." *Internet World Stats*. N.p., n.d. Web. 27 Nov. 2013.

"Internet Hall of Fame Innovator: Tim Berners-Lee." *Internet Hall of Fame*. N.p., n.d. Web. 27 Nov. 2013. <http://internethalloffame.org/inductees/tim-berners-lee>.

"JavaScript." *Mozilla Developer Network*. N.p., n.d. Web. 26 Nov. 2013. <https://developer.mozilla.org/en-US/docs/Web/JavaScript>.

Kohlbrenner, Eric. "The History of Virtual Machines." *Core of Information Technology*. N.p., n.d. Web. 25 Nov. 2013. <http://www.cs.gmu.edu/cne/itcore/virtualmachine/history.htm>.

Lucco, Steve. "TypeScript: Application Scale Development." Microsoft Research, 16 July 2013. Web. 27 Nov. 2013. <http://research.microsoft.com/en-us/events/fs2013/steve-lucco_modernprogramming.pdf>.

"The MEAN Stack: MongoDB, ExpressJS, AngularJS and Node.js." *The MongoDB Blog*. N.p., 30 Apr. 2013. Web. 27 Nov. 2013. <http://blog.mongodb.org/post/49262866911/the-mean-stack-mongodb-expressjs-angularjs-and>.

"Microsoft Internet Explorer 3.0 Beta Now Available." *Microsoft News Center*. N.p., 29 May 1996. Web. 27 Nov. 2013. <http://www.microsoft.com/en-us/news/press/1996/may96/ie3btapr.aspx>.

"Microsoft Plays Prank on Netscape after Bash." *Mozilla Stomps IE*. N.p., 2 Oct. 1997. Web. 27 Nov. 2013. <http://home.snafu.de/tilman/mozilla/stomps.html>.

Miller, Mark. ""Future of Javascript" Doc from Our Internal "Javascript Summit"" *GitHub Gists*. Google, n.d. Web. 24 Sept. 2013. <https://gist.github.com/paulmillr/1208618>.
"The New JavaScript Engine in Internet Explorer 9." *IEBlog*. N.p., 18 Mar. 2010. Web. 28 Nov. 2013. <http://blogs.msdn.com/b/ie/archive/2010/03/18/the-new-javascript-engine-in-internet-explorer-9.aspx>.
Niiler, Eric. "Netscape's IPO Anniversary and the Internet Boom." *Digital Life*. NPR. 9 Aug. 2005. Radio.
"Node.JS: The Good Parts? A Skeptic's View." *YouTube*. YouTube, 21 June 2013. Web. 27 Nov. 2013. <http://www.youtube.com/watch?v=CN0jTnSROsk>.
"The Original HTTP as Defined in 1991." *W3C*. W3C, n.d. Web. 27 Nov. 2013. <http://www.w3.org/Protocols/HTTP/AsImplemented.html>.
Paul, Ryan. "Firefox to Get Massive JavaScript Performance Boost." *Ars Technica*. N.p., 22 Aug. 2008. Web. 27 Nov. 2013. <http://arstechnica.com/information-technology/2008/08/firefox-to-get-massive-javascript-performance-boost/>.
"Quick History of Javascript by Crockford." *YouTube*. YouTube, 05 Sept. 2013. Web. 27 Nov. 2013. <http://www.youtube.com/watch?v=t7_5-XYrkqg>.
Rauschmayer, Axel. "A Brief History of ECMAScript Versions (including Harmony/ES.next)." *Web Builder Zone*. N.p., Mar. 2011. Web. 27 Nov. 2013. <http://css.dzone.com/articles/brief-history-ecmascript>.
Rauschmayer, Axel. *The Past, Present, and Future of JavaScript Where We've Been, Where We Are, and What Lies Ahead*. Sebastopol, CA: O'Reilly Media, 2012. Print.
"The RedMonk Programming Language Rankings: January 2013." *RedMonk*. N.p., n.d. Web. 27 Nov. 2013.
"A Re-introduction to JavaScript." *Mozilla Developer Network*. Mozilla, n.d. Web. 27 Nov. 2013. <https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript>.
Resig, John. "Asm.js: The JavaScript Compile Target." *Ejohn.org*. N.p., n.d. Web. 24 Sept. 2013. <http://ejohn.org/blog/asmjs-javascript-compile-target/>.
Severance, Charles. "JavaScript: Designing a Language in 10 Days." *IEEE Computer Society*. N.p., Feb. 2012. Web. 27 Nov. 2013. <http://www.computer.org/csdl/mags/co/2012/02/mco2012020007.html>.
Sheff, David. "Wired 8.08: Crank It Up." *Wired.com*. Conde Nast Digital, Aug. 2000. Web. 27 Nov. 2013. <http://www.wired.com/wired/archive/8.08/loudcloud.html?pg=1>.
Sink, Eric. "Memoirs From the Browser Wars." Web log post. *Eric Sink*. N.p., Apr. 2003. Web. 27 Nov. 2013. <http://www.ericsink.com/Browser_Wars.html>.
"Sir Timothy Berners-Lee Interview." *Academy of Achievement*. N.p., 22 July 2007. Web. 27 Nov. 2013. <http://www.achievement.org/autodoc/page/ber1int-1http://www.achievement.org/autodoc/page/ber1int-1>.
Stachowiak, Maciej. "Introducing SquirrelFish Extreme." *Surfing Safari*. N.p., 18 Sept. 2008. Web. 28 Nov. 2013. <https://www.webkit.org/blog/214/>.
"Strict Mode." *Mozilla Developer Network*. N.p., n.d. Web. 27 Nov. 2013. <https://developer.mozilla.org/en-

US/docs/Web/JavaScript/Reference/Functions_and_function_scope/Strict_mode>
.
Swartz, Aaron. "A Brief History of Ajax." *Raw Thought*. N.p., 22 Dec. 2005. Web. 27 Nov. 2013. <http://www.aaronsw.com/weblog/ajaxhistory>.
Takada, Mikito. "Understanding the Node.js Event Loop." *Mixus Tech Blog*. N.p., 1 Feb. 2011. Web. 27 Nov. 2013. <http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/>.
Trex, Ethan. "9 People, Places & Things That Changed Their Names." *Mental_floss Blog*. N.p., n.d. Web. 27 Nov. 2013. <http://blogs.static.mentalfloss.com/blogs/archives/22707.html>.
"TypeScript." *Welcome to TypeScript*. Microsoft, n.d. Web. 27 Nov. 2013. <http://www.typescriptlang.org/>.
"Unreal Engine 3 in Firefox with Asm.js." *YouTube*. YouTube, 02 May 2013. Web. 28 Nov. 2013. <http://www.youtube.com/watch?v=BV32Cs_CMqo>.
"Usage of JavaScript Libraries for Websites." *Web Technology Surveys*. W3Techs, n.d. Web. 27 Nov. 2013. <http://w3techs.com/technologies/overview/javascript_library/all>.
"V8: An Open Source JavaScript Engine." *YouTube*. YouTube, 15 Sept. 2008. Web. 27 Nov. 2013. <http://www.youtube.com/watch?v=hWhMKalEicY>.
"We Knew the Web Was Big..." *Google Official Blog*. Google, 25 July 2008. Web. 27 Nov. 2013. <http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html>.
Wiedermann, Ben, Vineeth Kashyap, John Sarracino, John Wagner, and Ben Hardekopf. "Type Refinement for Static Analysis of JavaScript." (2013): n. pag. Print.
"Windows 95 Opens Door for Microsoft's 50% Profit Jump." *Los Angeles Times*. Los Angeles Times, 23 July 1996. Web. 27 Nov. 2013. <http://articles.latimes.com/1996-07-23/business/fi-27151_1_fourth-quarter-profit>.
Zakai, Alon. "Big Web App? Compile It!" *Kripken.github.io*. Github, n.d. Web. 24 Sept. 2013. <http://kripken.github.io/mloc_emscripten_talk/>.