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 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 explosive 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. Any programmer worth his paycheck can tell you that 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 JavaScript was introduced in 1995. Like C is to Unix, JavaScript is 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 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] In fact, JavaScript has been called the world's most popular programming language. More important however, than how many people are writing JavaScript, is how many people

1 http://www.internetworldstats.com/emarketing.htm
2 http://redmonk.com/sogrady/2013/02/28/language-rankings-1-13/

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, JavaScript is surrounded by controversy and 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 complete novices. At the same time, JavaScript experts are pushing the language far beyond what people thought was possible. It is 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. JavaScript is ubiquitous and popular. Yet it is chastised and criticized. Is JavaScript a toy language? Or is it the 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 an excellent tool for describing documents; HyperText Transfer Protocol (HTTP) is an excellent protocol for fetching those documents. But the Web was not built for interacting with data and performing heavy computations. Today, however, the answers are murkier. Clever engineers have pushed the Web towards loftier goals. We now ask HTML to describe interactive tools and HTTP to manage the input and output of complex systems. JavaScript, a language designed to register browser events and tie together HTML components, has been pushed the furtherest: we ask it to define the behavior of increasingly complex applications.

The Web today is not just a series of documents strung together with links. The Web is a software platform—perhaps even the beginnings of a distributed operating system. Companies are

3 http://www.theguardian.com/technology/2013/feb/07/mobile-internet-outnumber-people

developing real software for the Web, and Web technologies are creeping into other platforms. Programmers are writing JavaScript as if it was 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 is not a toy.

The story of how JavaScript ended up in this situation is strange. Although perhaps we got lucky, and (arguably) the language is not as hard to use as originally thought, why did we choose to rely on JavaScript in the first place? Why take a chance on the construction of the most culturally central pieces of technologies we have made? The answer is that we did not choose to use JavaScript. The Web was not built or decided by anyone; it was built by everyone. The series of events that lead to JavaScript's current position within software development communities was truly a series of accidents. Political forces with a myriad of agendas have shaped the history, community and syntax of the language. Only looking back can we see why JavaScript has risen to its position. JavaScript, the most important language of the modern era, is the accidental king.

The future of the Web and the future of JavaScript are 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 (and maybe even HTML or HTTP) should be thrown out and we should start from scratch with a more careful plan. Others have argued JavaScript is the right language to move forward with, but it needs to be fixed in a big way. Others see no problem with the current trajectory of the Web: maybe we should accept that the requirements of these technologies are impossible to to predict or control. It is not clear which argument is right, or how these voices will shape the language or its platform. 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 mascot, a green foam creature named Mozilla, was standing proudly on the wreckage with a placard referring to recent market share percentages: “Netscape 72, Microsoft 18.”[7] The Browser Wars, which would ultimately decide which company brought 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 an HTML scripting language called Mocha. Eich is not quite sure which ten days it was. “From a calendar, I think it might have been May 6-15, 1995,” he writes in a 2013 answer to a question about Mocha on an online forum. The source files probably included a date —if anyone could find the originals—and also his office-mate at the time might remember, he adds.[8]

---

4 http://home.snafu.de/tilman/mozilla/stomps.html
5 http://home.snafu.de/tilman/mozilla/stomps.html
6 http://home.snafu.de/tilman/mozilla/stomps.html
7 http://home.snafu.de/tilman/mozilla/stomps.html
8 http://www.quora.com/JavaScript/In-which-10-days-of-May-did-Brendan-Eich-write-JavaScript-Mocha-in-1995

“That's the best I can do,” Eich apologizes.[9]

Browser Wars.

Unfortunately for those interested in the origin story of Mocha, those ten days in May do not seem any different from any other ten days of Navigator development 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 first wave of consumer Internet popularity with their hit product Navigator. Netscape had been born out of a small research project at the National Center for Supercomputing Applications (NCSA) at the University of Illinois at Urbana-Champaign. The research project, Mosiac, with development led by 22-year-old Marc Andreessen, was a tool for viewing Web content through a graphical interface. Although not the first tool created for such a purpose, Mosiac was the best. Seeing the potential for the project's commercial success, Andreessen left Illinois and took his expertise to Silicon Valley. There the young Andreessen met with Jim Clark, a Stanford professor who had made a fortune as a co-founder of Silicon Graphics. The two started Netscape Communications together. And almost immediately, their venture was a hit.

Web browsers at the time were a brand new phenomenon. Before the Web, graphical interfaces for the Internet had exclusively been closed proprietary systems, like America Online, which only allowed access to their own content. And before the graphical interface, accessing content on the Internet was a text-heavy activity, largely reserved for geeks and scientists. The World Wide Web was graphical *and* open, allowing anyone to create content for anyone else to view. The promise of this technology, with 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.[10] It was certainly the fastest growing software

9 http://www.quora.com/JavaScript/In-which-10-days-of-May-did-Brendan-Eich-write-JavaScript-Mocha-in-1995
10 http://www.npr.org/templates/story/story.php?storyId=4792365

company in history and many were calling it the fastest growing company ever. Clearly, the Web hit a nerve with the public. 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!”[11]

Unfortunately for Netscape, celebration would have to be short-lived. Competition was heating up. Seeing the potential of Andreessen's innovation, Microsoft had offered to buy Netscape in 1994.[12] 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 and they were on track to release Windows 95 that year. But the explosive growth at Netscape was worrisome for Microsoft. Netscape was calling their product, perhaps prematurely, a “distributed operating system,” a claim that if true would threaten Microsoft's core business.[13] In order to complete, in early 1995 Microsoft licensed Mosiac, the same research product that Andreessen had worked on, from Spyglass Inc (an offshoot of University of Illinois that was created to commercialize the work of NCSA).[14] By the summer of 1995, Microsoft was only months away from releasing what amounted to a thin varnish over the Mosaic codebase that they were calling Internet Explorer.[15] Although late to the browser game, Microsoft was serious about their new venture. Due to the company's extraordinary success in the personal computing space, Microsoft had amassed an elite team of software engineers and had 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

11 http://www.wired.com/wired/archive/8.08/loudcloud.html?pg=4
12 http://www.youtube.com/watch?v=Rj49rmc01Hs
13 http://www.computer.org/csdl/mags/co/2012/02/mco2012020007.html
14 http://www.ericsink.com/Browser_Wars.html
15 http://www.ericsink.com/Browser_Wars.html

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 necessarily obvious. But there were visionaries at Netscape and elsewhere who saw the need for such a technology and understood this was a place for a big 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.[16] “We aimed to provide a 'glue language' for the Web designers and part time programmers.”[17]

But even with support from Andreessen, not everyone at Netscape was convinced that a glue language 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 (given its hardware agnostic nature). 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? The big debate inside Netscape became “why not just Java?”[18]

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 both. But in order to prove this idea to the company, Eich had to move quickly. Political forces at Netscape, with Microsoft squarely on their minds, demanded a proof of concept. “Doing this work so fast was important,”[19] Eich recalls in a talk at a conference on JavaScript in 2012. “We knew Microsoft was coming after us.” Eich wrote the Mocha interpreter in ten days because the fate of Mocha was not

16 http://www.computerworld.com.au/article/255293/a-z_programming_languages_javascript/
17 http://www.computerworld.com.au/article/255293/a-z_programming_languages_javascript/
18 https://brendaneich.com/2008/04/popularity/
19 http://www.youtube.com/watch?v=Rj49rmc01Hs

sealed. Until Eich could point at something real, Andreessen's arguments for the language had no ammo. “I had to [write the interpreter] for an internal demo, because otherwise people would doubt that it was either real or necessary,” Eich recalls at the 2012 conference.[20] Apparently his demo was impressive. Netscape shipped Mocha, by then renamed JavaScript, with Navigator 2.0 in December 1995.

Sun and Netscape.

By the time Navigator 2.0 shipped, Eich's language had been renamed twice. First to “LiveScript” in September 1995, to jell with the rest of Netscape's products (which all began with a “Live” prefix). Then subsequently 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 language 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, as part of the deal Netscape agreed to rename the language “JavaScript.”

Over the years, the marketing decision behind the name “JavaScript” has led to a lot of confusion. Because Java was so well regarded at the time, the confusion was probably a good thing for Netscape in JavaScript's early years. But even today software developers and laypersons alike have been known to 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.[21] In reality the languages are fundamentally different. Although they share some superficial similarities, the cores of each language have almost nothing to do with each other. That said, the politics of Java direct impacted JavaScript, before, during and after JavaScript's development.

20 http://www.youtube.com/watch?v=Rj49rmc01Hs
21 http://www.java.com/en/download/faq/java_javascript.xml

Java started its life in 1991 as a language called Oak, written for a research project developed by a small team at Sun that was very 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.”[22] 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 software engineer on the Green Team wrote Oak.[23]

Gosling's 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.[24] As such, Oak allowed Green Team engineers to write a program once and then run that program on many different devices without any extra work. 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 technology.[25] It was not clear whether they were making a ton of progress or that Oak would ever see the light of day.

But in 1995, the world was changing rapidly. Suddenly 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. James Gosling explains their 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

22 http://web.archive.org/web/20050420081440/http://java.sun.com/features/1998/05/birthday.html
23 http://www.cs.gmu.edu/cne/itcore/virtualmachine/history.htm
24 http://web.archive.org/web/20050420081440/http://java.sun.com/features/1998/05/birthday.html
25 http://web.archive.org/web/20050420081440/http://java.sun.com/features/1998/05/birthday.html

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."[26] The processor independent nature of Java, which was not easy to replicate in C++ or another serious programming language 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. So that's what we did."[27]

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.[28] Although Java had been developed separately from concerns associated with the Web, it appeared to the world that Java was built for Navigator. This is exactly what Sun and Netscape executives wanted. The partnership was a strategic play that they knew could forward the momentum of both Sun and Netscape. Both technologies, Java and Navigator, were receiving a lot of press and enthusiasm. Together, could they challenge Microsoft? More pessimistically, if they did not act together, did they stand a chance? Douglas Crockford, a major figure in the JavaScript development community, said in 2011 that he thought 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."[29]

Java and JavaScript.

The close relationship between Sun and Netscape in 1995 cast a long shadow over the development of JavaScript. According to Eich, he was hired at Netscape to embed a functional language in Navigator. As Eich writes is a 2008 blog post: "As I've often said, and as others at

26 http://web.archive.org/web/20050420081440/http://java.sun.com/features/1998/05/birthday.html
27 http://web.archive.org/web/20050420081440/http://java.sun.com/features/1998/05/birthday.html
28 http://web.archive.org/web/20050420081440/http://java.sun.com/features/1998/05/birthday.html
29 http://www.youtube.com/watch?v=t7_5-XYrkqg

Netscape can confirm, I was recruited to Netscape with the promise of 'doing Scheme' in the browser... Whether that language should be Scheme was an open question, but Scheme was the bait I went for in joining Netscape."[30] 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.'"[31] 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, asking that the language look like Java, Eich notes, "ruled out Perl, Python, and Tcl, along with Scheme," which otherwise would have been reasonable choices to adapt for Navigator.[32] In fact, there was only a certain class of languages that looked like Java: languages in the C-family.

Java's syntax was designed to look like C++, which was Java's main inspiration and competition. The C-family of languages has a very particular (and popular) brace syntax. However this style was designed for procedural programming and none of the popular scripting languages (which largely took inspiration from the LISP-family of languages) used braces to offset functions or blocks of code. Of course, Eich could not simply use Java. Java's rigid class system, static typing and lack of functional capabilities made it attractive for large-scale application development, but unreasonable for writing short scripts. In order to satisfy these 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 to 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."[33]

30 https://brendaneich.com/2008/04/popularity/
31 https://brendaneich.com/2008/04/popularity/
32 https://brendaneich.com/2008/04/popularity/
33 http://www.computer.org/csdl/mags/co/2012/02/mco2012020007.html

The Good, the Bad and the Ugly.

Given all that surrounded the origins of JavaScript, and the requirements that political forces 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.[34] 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 has become known as the 'DOM': APIs from JS to control windows, documents, forms, links, images, etc., and to respond to events and run code from timers,” said Eich in the ComputerWorld interview.[35] Without the DOM interface, JavaScript was of little interest to anyone at Netscape. So the design of the language itself had to be 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 stop. JavaScript's functional concepts are taken from 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. For the most part, everything in JavaScript is an object. Even arrays, which are primitives in many languages, are objects in JavaScript. However, unlike

34 http://www.computerworld.com.au/article/255293/a-z_programming_languages_javascript/
35 http://www.computerworld.com.au/article/255293/a-z_programming_languages_javascript/

essentially ever other popular object-oriented language, JavaScript does not support classes. In Java, a class based object-oriented language, classes representing data and functions (for operating on that data) are written by the programmer. Then instances of these classes (objects) are created and used at runtime. On the other hand, in JavaScript there are no classes. JavaScript objects can be polymorphic, and a programmer can encapsulate data within a JavaScript object, but inheritance is nothing like Java's clean implementation. To create a new object (say a Racecar object) by extending an object that already exists (say a Car object), a programmer must “prototype” the object and add new functionality. This copying happens at runtime, rather than at compile time and means that clean data encapsulation inside the copied object is sometimes not possible.

JavaScript's type system is also poorly designed. For instance, there is only one “Number” type in JavaScript.[36] 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 floating point numbers under the 64-bit precision IEEE standard. This leads to unfortunate consequences:

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

Dealing with very large or very small numbers (numbers that lack reasonable precision in the IEEE standard) is difficult. It is hard to imagine a bank using JavaScript primitive numbers to keep track of interest payments.

Type conversation and comparison in JavaScript is also a common pain-point. 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:

36 http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf

```
> '0' == 0
true
> 0 == ''
true
> '0' != ''
true
> '' == false
true
```

JavaScript has the values null, undefined, NaN (“not a number”), Infinity and false, which are all used in different contexts to signify that something is a “bad” value that cannot be operated on in a meaningful way. Each of these has slightly different meaning, but for beginners as well as seasoned programmers, those meanings are not always obvious. Testing equality among these “bad” values can be difficult. To make things extra confusing, NaN does not equal itself:

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

There are long list of things that make JavaScript a difficult language to work with. For instance, JavaScript uses function scope rather than block scope. Variables can be declared in the global scope of a program from any scope if a programmer forgets a “var” declaration. The language does not have any built in module system, rather it was designed to rely on HTML's “<script>” tag in order to interface with other JavaScript code. Lastly, the language has no concept of input or output. It completely relies on the host environment to provide mechanisms for communicating with the outside world.[37] In short: as it was written, JavaScript was good for writing short scripts, and that's it. Although Eich's work was impressive given the circumstances, he did not produce a Java-quality language.

37 https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

JavaScript in 1995 was a toy language.

Microsoft crushes Netscape.

On August 13, 1996, Microsoft released Internet Explorer 3. All around, the application was a better product than its predecessors. IE3 no longer relied on the Mosaic codebase and it included the first market implementation of Cascading Style Sheets (CSS). With IE3, Microsoft finally approached feature parity with Navigator. Before it was superseded by Internet Explorer 4, the browser took significant market share from its competition. In 1996, Microsoft's market share of the space was in the single digits. By the end of 1997, it was closer to 30%.[38]

Among a multitude of improvements that the browser brought over its predecessors, Internet Explorer 3 shipped with JavaScript support. Microsoft had reversed engineered Navigator's JavaScript with an implementation they called JScript (for copyright reasons).[39] 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 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 more and more market share from NetScape. Microsoft had a number of market advantages that made their progress much easier. The most conspicuous was financial resources. Enormous profits from Microsoft's other businesses, $559 million in the second quarter of 1996, allowed Microsoft to give away Internet Explorer for free and market it heavily.[40] NetScape could not afford to compete in this way. Microsoft's second big advantage

38 http://www.cnn.com/TECH/computing/9810/08/browser.idg/
39 http://www.microsoft.com/en-us/news/press/1996/may96/ie3btapr.aspx
40 http://articles.latimes.com/1996-07-23/business/fi-27151_1_fourth-quarter-profit

was its domination of the operating system space. Windows faced no real competition. Taking advantage of this, Microsoft began bundling Internet Explorer with its operating system. 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 users had nothing to compare Microsoft's browser against; it is possible that many did not even 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. 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, Microsoft had over 90% market share. Netscape was acquired by AOL.

Although its original implementation did not become its dominant form, JavaScript lived on. NetScape had submitted JavaScript to a standards board for a language standardization in 1996, and more practically Internet Explorer implemented the language. When other browsers came along later and took market share away from Internet Explorer, implementing JavaScript for the makers of these browsers was a no-brainer. Mozilla (a non-profit born out of the rubble of Netscape) introduced Firefox, Apple introduced Safari and Google introduced Chrome. All three of these browsers supported JavaScript from the 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 browser different scripting language would be incredibly difficult. The question is no longer up for debate.

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."[41] 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 become 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.'"[42] 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.[43] As the Web grew however, the need for link aggregators like Yahoo! to provide content organization as a service became increasing 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 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

41 http://www.theguardian.com/business/2008/feb/01/microsoft.technology
42 http://blogs.static.mentalfloss.com/blogs/archives/22707.html
43 http://visual.ly/brief-history-web-standards

structure," Larry Page, one of these students, told Wired Magazine in 2008.[44] The Web, Page and his partner (Sergey Brin) theorized, was the largest graph ever created.[45] 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. For instance, if all the best webpages on dogs linked back to a certain page, that page likely contained some high value information on dogs. 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. Google, the search engine built around their algorithm, blew every competitor out of the water.

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, Google's announced that their index had recorded over a trillion unique pages.[46]

The number of Internet users moved in tandem with increasing content. In December 1995, the month Netscape launched Navigator 2.0, there were estimated to be 16 million Internet users in the world. In 2013, there are 2.7 billion.[47] Such dramatic changes in the usage of the Internet caused major changes in the nature of the technology. Once a provence of static information for research scientists, the Internet's new 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 little language called JavaScript.

---

44 http://www.wired.com/wired/archive/13.08/battelle.html
45 http://www.wired.com/wired/archive/13.08/battelle.html
46 http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html
47 http://www.internetworldstats.com/emarketing.htm

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. Java applets were one option. Thanks to the work of Netscape and Sun, a developer could compile a Java program into an applet and embed it into a webpage. Unfortunately, applets were clunky; interaction was limited and the source code was compiled, so it was necessarily hidden from the browser. A Java applet had no good way to know about the HTML it was embedded in. Another popular solution, especially among game developers, was Flash. Flash, developed by Macromedia (and then Adobe) opened up some very serious animation capabilities, but it presented a lot of the same issues that Java did: the power of the software was sometimes more of a hinderance than a help. Tim Berners-Lee, the CERN Internet pioneer, has referred to this problem as Principle of Least Power. He argues in a 2013 article that simpler Web technologies are typically the best: "If, for example, a web page with weather data has [a simple data format] describing that data, a user can retrieve it as a table, perhaps average it, plot it, deduce things from it in combination with other information." More powerful technologies, he continues, with hidden implementations, give users less power and are therefore less desirable: "At the other end of the scale is the weather information portrayed by the cunning Java applet. While this might allow a very cool user interface, it cannot be analyzed at all. The search engine finding the page will have no idea of what the data is or what it is about. The only way to find out what a Java applet means is to set it running in front of a person."[48]

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

48 http://www.w3.org/DesignIssues/Principles.html

access to browser APIs, the HTML it was embedded in and the styling of that HTML. Although JavaScript was arguably much less powerful than its competitors 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 different 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++ and Java, writing JavaScript, which lacked a module system and a reasonable type system, was not ideal. The language developed a reputation as a “toy” language. Nevertheless, as the world came online and traffic rewarded webpages with more complexity and interaction, developers had no choice. JavaScript had to be written. As the Web exploded, the amount of JavaScript being written and the number of people who were writing it grew rapidly as well. These two forces created enormous pressure to improve the language. 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 development were actually problems with browsers, not problems with the language itself. In particular, standards fragmentation and the underdevelopment of browsers deserve some of the 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.[49] Fierce competition had pushed the two companies to

49 http://www.webstandards.org/about/history/

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.[50] Browser inconsistencies meant that developing for the Web became increasingly difficult and expensive in the late 90s.

Although standards bodies like W3C probably deserve some credit for improving the coherency of Web technologies, most of the problems resulting from browser inconsistencies were fixed as side-effects of larger trends. By 2000, Microsoft (with arguably anti-competitive business practices) had all but put Netscape out of business. The problems of browser fragmentation vanished overnight: even if Microsoft wanted to ignore W3C's recommendations, the mystery of what was supported for which users disappeared. In addition to this, Internet Explorer (and its new competitors, when they arrived) matured into better software over time. Fewer browser bugs, better built-in tools and more consistent 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 can 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.[51] Douglas Crockford, a well known JavaScript developer and somewhat 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.' But W3C had been waiting a long time for an opportunity to tell Netscape to 'go to hell.'”[52] Netscape could not take no for an answer however. Microsoft's adoption of JScript had made it

50 https://developer.mozilla.org/en-US/docs/Web/HTML/Element/blink
51 The Past Present and Future of JavaScript (Rauschmayer) 5
52 http://www.youtube.com/watch?v=t7_5-XYrkqg

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.[53] Ecma was, as Crockford notes, “a long way to go for a California software company.”[54] 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.”[55] 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.[56]

Ecma has improved the language incrementally, but significantly over the years. 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, new control statements, try/catch exception handling, tighter definition of errors, and formatting for numeric output.”[57] The ECMAScript 3 standard was incorporated into both the JavaScript and JScript implementations soon after it was released.

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.[58] ECMAScript 5, the more incremental release, was agreed upon in

---

53 http://www.ecma-international.org/
54 http://www.youtube.com/watch?v=t7_5-XYrkqg
55 https://mail.mozilla.org/pipermail/es-discuss/2006-October/000133.html
56 http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
57 http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf
58 The Past Present and Future of JavaScript (Rauschmayer) 10

December 2009.[59] 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 its impossible to assign variables at the global scope.[60] 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 a version of ECMAScript 5.[61]

Although Ecma's changes are significant, and have improved the core of the language significantly, the true forces pushing JavaScript development forward have been outside the standards 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, the British computer scientist, is credited with inventing the World Wide Web. 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.[62] 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

59 http://css.dzone.com/articles/brief-history-ecmascript
60 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/Strict_mode
61 https://developer.mozilla.org/en-US/docs/Web/JavaScript
62 http://www.achievement.org/autodoc/page/ber1int-1

wrote the first web client and server in 1990.[63]

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 severely limited. Berners-Lee could not have anticipated the significance of his invention or 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.[64] And although subsequent version of HTTP, including HTTP 1.1 which 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 needs of the real 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. A client and a server could have an ongoing *relationship*. Although it is hard to say for sure who invented the Web application, the Internet entrepreneur and venture capitalist Paul Graham claims that his product, Viaweb, created in 1995, was the first Web application that allowed persistent user data.[65]

Web applications in the late 90s were smart, but they were limited. HTTP 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 already been enjoying for many years. Unless a webpage was refreshed, changes in the state of the server code could not be reflected in

63 http://internethalloffame.org/inductees/tim-berners-lee
64 http://www.w3.org/Protocols/HTTP/AsImplemented.html
65 http://www.paulgraham.com/first.html

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 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. JavaScript was completely sandboxed to the browser.

Ajax Revolution.

The limitations of HTTP were a massive problem for web developers. At the same time that desktop applications like Adobe Photoshop and Microsoft Excel were becoming the standard tools for entire industries, no one could write a decent email application for the Web. Fortunately, a solution emerged. In the late 90s, a software engineer named Alex Hoppman came up with a way to transfer data back and forth from a server without refreshing a webpage.

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].”[66] Unfortunately, the team doing the dynamic pages was having trouble doing server communication in a reasonable way. “They were basically doing hacky form-posts back to the server,” says Hoppman. Out of necessity and curiosity, Hoppman implemented a new solution: “That weekend I started up Visual Studio and whipped up the first version of what would become XMLHTTP.”[67]

Hoppman's XMLHTTP allowed JavaScript code to make GET and POST requests to a back-end server and perform data transactions *without* refreshing the page. Although it did not seem

66 http://www.alexhopmann.com/xmlhttp.htm
67 http://www.alexhopmann.com/xmlhttp.htm

revolutionary at the time, Hoppman's weekend project was a sea change for Web application development. More immediately for Hoppman and Microsoft, XMLHTTP meant that new emails coming in to Outlook Web Access could be *pulled down* by the JavaScript on the page and then rendered for the user.

The XMLHttpRequest object for JavaScript, providing an interface to this technology, was first introduced with Internet Explorer 5.[68] The library and the common best practices associated with it are now collectively called Ajax ("Asynchronous JavaScript and XML.") Arron Swartz, the late Web programmer and Internet activist, has written 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."[69] Although no one was using it at first, Ajax quietly changed all the basic assumptions that developers had made about HTTP. As Swartz says, "XMLHttpRequest allowed the JavaScript inside web pages to do something they could never really do before: get more data."[70]

Supercharged JavaScript.

As Ajax methodologies became popular, and as Web users continued to demand more complex experiences on the Web, there was increased interest in building good tools for JavaScript. One effect of this was the introduction of good JavaScript frameworks that made common Web development tasks easy. Far and away, the most popular of these frameworks 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).[71] JQuery, introduced in 2005, adds intuitive Web-specific idioms, including built-in Ajax, CSS and JSON support. For example, the following code, which would

68 http://www.alexhopmann.com/xmlhttp.htm
69 http://www.aaronsw.com/weblog/ajaxhistory
70 http://www.aaronsw.com/weblog/ajaxhistory
71 http://w3techs.com/technologies/overview/javascript_library/all

be far more verbose in “plain 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);
  });
});
```

Another effect of increased interest in JavaScript development was that JavaScript performance suddenly became important. When JavaScript use was limited to a few lines here and there to tie together Web components, the speed of that code's execution was not of particular concern. However, by the mid-2000s, with the power of Ajax techniques and expressive power of JQuery under their belts, developers were building larger and larger pieces of software in JavaScript. Web applications were starting to test the performance limits of the JavaScript interpreters that existed at the time. Google was leading the way with Web application complexity, and therefore was facing performance bottlenecks years ahead of other developers. In order to facilitate the complex interfaces of their Web applications, Google was creating their own proprietary JavaScript frameworks and using Ajax techniques extensively. For instance, when a user drags the Google Maps interface around, it automatically fetches new map data. The result is that the user feels like they can zoom and pan around petabytes of satellite data as if it was stored on their local disk. Such an effect is only possible with 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.[72]

The development of the Chrome browser, which began in 2006, was a strategic move on Google's part to push the browser market forwards. The quality of the Web browsing experience was central to their business and the company wanted the browsers to innovate. Peter Kasting, a Chrome

72 http://blogs.wsj.com/digits/2009/07/09/sun-valley-schmidt-didnt-want-to-build-chrome-initially-he-says/

engineer, said in 2011 that "...the primary goal of Chrome is to make the web advance as much and as quickly as possible. That's it." According to Kasting, "It's completely irrelevant whether Chrome actually gains tons of users or whether instead the web advances because the other browser vendors step up their game and produce far better browsers. Either way the web gets better."[73] The key metric for improving the Web that Google seemed to be targeting was JavaScript speed. Chrome's JavaScript engine, named "V8" after the type of car engine, was designed to be fast. Google's introduction to V8 makes that clear: "JavaScript programs have grown from a few lines to several hundred kilobytes of source code... V8 is a new JavaScript engine specifically designed for fast execution of large JavaScript applications."[74]

The lead developer on the V8 project, Lars Bak, has said that the main purpose of V8 "is to raise the performance bar of JavaScript out there, in the marketplace."[75] V8 has been successful in this stated goal.[76] Mozilla, with Brendan Eich as CTO, made major improvements to its SpiderMonkey engine around the same time that V8 was released. SpiderMonkey, the JavaScript engine in Mozilla's Firefox browser, was a refactor of the Mocha codebase originally written by Eich in 1996.[77] The engine remained largely unchanged for years until it was completely revamped for Mozilla's Firefox 3.5 release: Firefox 3.5 included the "TraceMonkey" engine, which implementation just-in-time compilation for JavaScript.[78] Although Mozilla probably would not admit that pressure from Google was part of their decision to develop TraceMonkey, it probably had some effect. TraceMonkey was up to 40 times faster than Firefox 3's SpiderMonkey, and it was about on par with V8.[79]

Mozilla seemed to agree with Google on the growing importance of JavaScript performance. In a 2008 interview with ArsTechnica about TraceMonkey, Brendan Eich said that Mozilla wanted to "get

73 http://www.tomshardware.com/news/google-chrome-web-browser-mozilla-firefox,14378.html
74 https://developers.google.com/v8/intro
75 http://www.youtube.com/watch?v=hWhMKalEicY
76 https://developers.google.com/v8/design
77 https://brendaneich.com/2011/06/new-javascript-engine-module-owner/
78 http://arstechnica.com/information-technology/2008/08/firefox-to-get-massive-javascript-performance-boost/
79 http://arstechnica.com/information-technology/2008/08/firefox-to-get-massive-javascript-performance-boost/

people thinking about JavaScript as a more general-purpose language."[80] On his personal blog, he writes: "Once we had launched TraceMonkey, and Apple had launched SquirrelFish Extreme, the world had multiple proofs along with the V8 release that JS was no longer consigned to be 'slow' or 'a toy'"[81] In a few years, JavaScript performance experienced an order of magnitude increase.

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.[82] In order to support himself, Dahl picked up programming and wrote PHP Web applications. At the time, the Ruby on Rails framework, which allowed developers to write Web servers in Ruby, was very new. Dahl became interested in Rails, but his interest quickly turned to annoyance. Unfortunately, Rails was slow. Dahl recounts thinking about the framework a lot during this time in a 2011 interview: "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... It had zero concurrency."[83]

Out of frustration with Ruby on Rails, Dahl began work on a project to write a non-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."[84] Ruby, like Python and other interpreted languages, has a global interpreter lock, meaning it is not possible to work on 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."[85]

There were a few reasons that Dahl thought that JavaScript was the perfect candidate for writing

80 http://arstechnica.com/information-technology/2008/08/firefox-to-get-massive-javascript-performance-boost/
81 https://brendaneich.com/2011/06/new-javascript-engine-module-owner/
82 http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/
83 http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/
84 http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/
85 http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/

what eventually became known as 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 major players in the field 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.”[86]

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 notions of what it meant to meant to be a server-side JavaScript program.”[87] There is no concept of input and output in JavaScript, let alone the concepts of interfacing with the filesystem 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 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.”[88] JavaScript allowed Dahl to start from scratch. He wrote all all the building blocks of his JavaScript implementation so that they were non-blocking and could be run on Node's event-driven system.

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

86 http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/
87 http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/
88 http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/

```
var circle = require('./circle.js')
console.log('The area of a radius 4 circle is' + circle.area(4));
```

Although it is built for writing Web servers, Node opens up the entire operating system for JavaScript programs. The project has carved out a vast new territory for JavaScript, and it gives the language the tools to operate inside it. Therefore 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 and its capabilities have been changed immeasurably.

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

JavaScript has come along way since 1995. In 2013, the language boasts libraries and tools that make it 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 developments 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 Web developer in 2013 consider JavaScript as a reasonable 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 who had an idea for an application written completely in JavaScript. The Exavault team, based on Oakland, CA builds Web applications to help businesses share large files internally using the FTP protocol. For three months, Exavault's five person development team joined me one day a week to build NextFive, a collaboration tool written entirely in JavaScript. By the end of my three months with the team, we were able to launch 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 in increasingly common in the modern world, especially among software development teams. Advances in version control systems for code management and online meeting tools like Skype have allowed

teams to work together from separate parts of the world. But an online-only office environment sometimes causes inefficiencies. How can a team member keep track of what everyone else is working on?

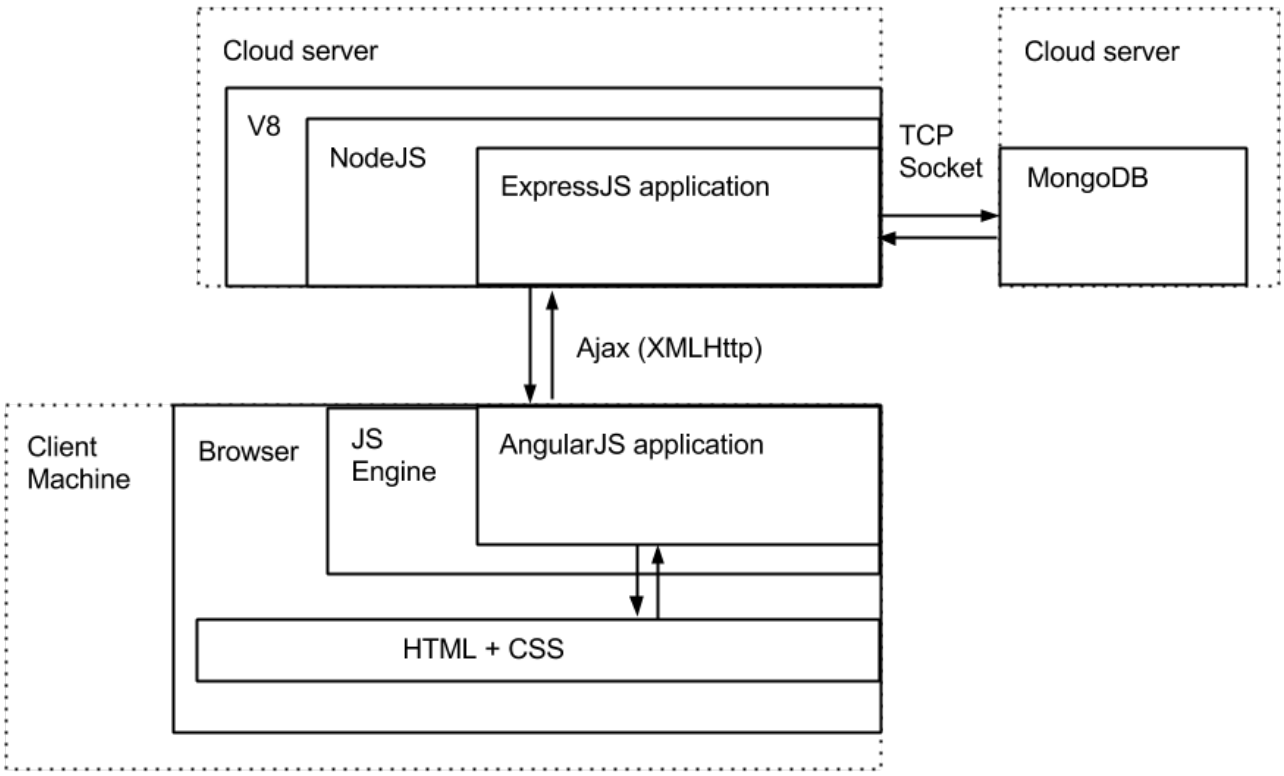
Exavault solves this problem with email reports. At the beginning of each week, every employee writes a short report detailing what she plans to accomplish each day that week and shares it with the rest of the team. The employee's manager and colleagues then have a chance to review her report and get a sense for what she will be working on. This increases the chances that inconsistencies will be caught early and the team's workload for the week will be known to all. 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 of time. In order to accomplish its goals, NextFive had to support secure login for employees, email integration, and interfaces for writing, reading and comparing reports among other features. In September we set out to build a beta version of NextFive with this feature set in three months. And of course, we did it all in JavaScript.

Lean and MEAN.

After a number of technical discussions in the first two weeks of the project, we eventually settled on a series of JavaScript technologies that together have been called the “MEAN” stack.[89] MEAN stands for MongoDB, ExpressJS, AngularJS and NodeJS. MongoDB is a database that allows for handling data in JSON, ExpressJS is a Web framework for NodeJS that 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 it 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 stack

89 http://blog.mongodb.org/post/49262866911/the-mean-stack-mongodb-expressjs-angularjs-and

has become a popular choice for writing full-stack 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 backend Express application. In turn, Express, which is running on NodeJS on a cloud server, is responsible for responding to Angular's requests by creating, updating or fetching data in the Mongo database over a socket connection. Although Express is responsible for handling user authentication and serving the Angular application to the browser, once the application has been initialized, the Angular application mostly drives the show. Express acts as an interface for Angular to interact with backend data through HTTP requests, rather than as the main center of application logic. This is a very different way of thinking about a Web application, and it speaks to the power of Angular.

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. We managed these modules and their dependencies with Node's built in package manager, NPM. We also used a JavaScript build tool, GruntJS, to automate build processes. For instance we configured Grunt to run all of our JavaScript code through a JavaScript linting program before starting the server each time.

Challenges.

While it might be 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, Google's official documentation for Angular is lacking; for a heavyweight, opinionated 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 pure JavaScript way of doing things. The Exavault team uses PHP for their main projects, so there was a significant amount of adjusting to write Node on the backend. For example, Node is designed so that all program input and output is done asynchronously. This means that the program does no waiting for program input and that all uses of input must be handled in a callback function. This is an entirely different way of reasoning through code. For example, the following Python code connects to a database and performs a simple query:

```
import sqlite3

db_cursor = sqlite3.connect('test.db').cursor()
db_cursor.execute('SELECT SQLITE_VERSION()')
data = db_cursor.fetchone()
```

```
## By the time we reach this line, the database query is complete.
print "SQLite version: %s" % data
```

This code 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.[90] 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.[91] When the Python program is executing the database query, on line 5, the program is completely halted. The program does not execute the print statement on the final line until the database query is done and the 'data' variable has been assigned. As it turns out, 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 sits idle in order to make that abstraction possible. From a human perspective, a billion CPU cycles is a fraction of a second, but in reality 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 code expression is not the same as assigning a variable to the results of a database query in Node. For instance, the following NodeJS code does not do what the Python code above did:

```
var user = User.findOne({name: "Brendan Eich"});
console.log(user); // This will print nonsense.
```

Instead we have to push a callback onto Node's event loop that will be called when the query returns.

90 http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/
91 http://blog.mixu.net/2011/02/01/understanding-the-node-js-event-loop/

We can only rely on the return value of the query being define *within* the callback. In this case, we define the callback anonymously within the call to findOne:

```
User.findOne({name: "Brendan Eich"}, function(err, user) {
  console.log(user);
}
// The database query is still executing at this point.
console.log("This will get printed before the database result.");
```

This kind of programming is everywhere in NodeJS and it can lead to some nasty looking code, where anonymous functions are declared within anonymous functions many levels deep. Fortunately, this is a little bit like what Ajax calls look like in browser-side JavaScript, so there is some precedence for it for JavaScript programmers. But 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.

Although we faced some development challenges, the advantages of a pure JavaScript implementation were obvious throughout the development process. Using one language for the entire stack was a huge 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 star a NextFive report, 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. So in a Ruby stack for instance, 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 the backend. But in a pure JavaScript application, the language is the same

throughout. This means 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 the browser and the server 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. For other features, the backend just sends raw data down the line and Angular handles the computation. Being able to push logic to both sides of the stack, without straining the brains of the programmers, is a noticeable advantage of a pure language stack.

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 zero extra interface code, and finally the data object is rendered in HTML with Angular's JSON notation. JSON data at every level means we can think of the data in the same way at every level. 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 deal with it.

Lastly, the modern tools for pure JavaScript development is are high quality. The popular Web frameworks for PHP, Python, and Ruby are all several years old. This means that they have worked out the bugs and people have used them to build real products, but they do not necessarily use the most advanced solutions. On the other hand, the tools for using a pure JavaScript stack tend to solve more contemporary problems with the best technology available. The best example of this effect for us was Angular. Before beginning development in September, we briefly considered using BackboneJS, a more battle-tested JavaScript browser framework. But in retrospect, the newer Angular framework was the right choice. In comparison to Angular, Backbone is minimalistic. It provides structure for a JavaScript application, but with fewer conveniences. For instance, with no extra code, Angular

supports two-way data binding between data models and views. What this means is that a data object rendered in HTML is connected to the JavaScript object that represents the same data. When the JavaScript object is changed, the HTML view updates automatically, and vice versa. This functionality is extremely powerful. It allows us to abstract the annoying parts of HTML away from the programmer and build more powerful user interactions. This kind of functionality does not exist in JavaScript frameworks built only a few years ago, so by using the newest components, we got a big advantage.

NextFive and JavaScript.

Unfortunately, NextFive does not test JavaScript's ability to scale to hundreds of thousands of users, hundreds of thousands of lines of code or hundreds of programmers. That kind of large-scale JavaScript development was not possible to test. However, NextFive is a 2500 line JavaScript application spanning over 35 JavaScript files that relies on tens of thousands of lines of JavaScript modules. The application was built over a significant period of time by 6 programers. So it seems like a large enough study of pure JavaScript development that we can reasonably make conclusions about *medium-scale* JavaScript development.

Overall, NextFive has shown that pure JavaScript development with the MEAN stack is comparable to development with stacks in Python, Ruby, PHP or other dynamically typed languages. JavaScript on the server is newer and arguably more bug prone, but is faster than Ruby or Python because of Node's event loop. JavaScript applications benefit from the same language and data representation at all levels. Ruby and Python benefit from larger Web communities and better documentation. There is no question that pure JavaScript development is different than programming the same application in Ruby or Python, but after writing a significant application in pure JavaScript, it does not seem like choosing one or the other for backend programming is an obvious choice. In 2013, pure JavaScript development is a reasonable choice. It is clear that 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

Modern JavaScript is leaps and bounds better than the language Brendan Eich wrote in ten days. 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.[92]

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. The 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 bent by Google, Microsoft and Mozilla, some of the heaviest and more influential users of the language.

Lingering problems.

Despite its successes, JavaScript remains problematic. Many of the design problems present in 1995 still exist: 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

92 http://www.youtube.com/watch?v=CN0jTnSROsk

JavaScript. In modern JavaScript, the equality and plus operators still do confusing auto type-conversion:

```
> 1+1+1
3
> 1+1+'1'
'21'
> '1'+1+1
'111'
```

For simple scripts, this 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 developers are demanding more and more from JavaScript. JavaScript applications are becoming big enough that one person can not reasonably understand them. 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. Attempts to do some level of static analysis on dynamic languages is an open area of research in Computer Science, but results have been limited.[93] If the Web is going to keep growing, and allow support for increasingly complex applications that can 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

93 Widermann, Type Analysis for the Static Analysis of JavaScript

scale.[94]

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.[95] 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.”[96] Microsoft is making an effort to make TypeScript as attractive as possible for JavaScript developers. For instance, TypeScript and its compiled output are designed to be as readable as possible for JavaScript developers. TypeScript's marketers boast that “The output of the TypeScript compiler is idiomatic JavaScript.”[97]

Although Microsoft wants TypeScript to read and write like JavaScript, their 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.”[98] 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.”[99] TypeScript is Microsoft's effort to introduce large application development concepts to JavaScript.

TypeScript is designed for programming in the large.[100] In contrast to other attempts to

94 http://wiki.ecmascript.org/doku.php?id=harmony:specification_drafts
95 http://www.typescriptlang.org/
96 http://www.typescriptlang.org/
97 http://arstechnica.com/information-technology/2012/10/microsoft-typescript-the-javascript-we-need-or-a-solution-looking-for-a-problem/
98 http://channel9.msdn.com/Events/Build/2013/3-314
99 http://channel9.msdn.com/Events/Build/2013/3-314
100 http://research.microsoft.com/en-us/events/fs2013/steve-lucco_modernprogramming.pdf

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 JavaScript and Java, will find TypeScript familiar:

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

TypeScript is (optionally) statically typed and supports classes, modules and interfaces among other modern language concepts. These additions, especially the inclusion of static types, mean 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.”[101] 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. Microsoft also hopes that TypeScript easy enough for JavaScript developers to learn that the language can be used today.

Abstraction and avoidance.

At the core of the TypeScript project is the idea that JavaScript can be abstracted away from the

101 http://channel9.msdn.com/Events/Build/2013/3-314

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 Github.[102]

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."[103]

```
square = (x) -> x * x
alert "that exists" if this_exists?


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

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

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

}).call(this);
```

Brendan Eich, the CTO of Mozilla, has said that CoffeeScript influences how he thinks about JavaScript.[104] And Eich is (to some extent) a supporter of the core philosophy behind both TypeScript

102 http://readwrite.com/2012/02/14/redmonk-programming-language-r#awesm=~onvvKt678lJGD5
103 http://coffeescript.org/
104 https://brendaneich.com/tag/javascript-ecmascript-harmony-coffeescript/

and CoffeeScript: one of his goals for ECMAScript 6 is for the language to be “a better language for writing code generators.”[105] 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 (very) 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.”[106] 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).

```
// 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 is much lower level that traditional JavaScript programs. It also means that ASM.js programs analyzed statically. ASM.js is

105https://brendaneich.com/brendaneich_content/uploads/ES.003.png
106 http://asmjs.org/faq.html

just JavaScript, so it can be interpreted by any modern JavaScript interpreter. But it does not *have to be interpreted*. 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 completely.[107] Mozilla's approach to the future of the Web, besides continuing to evolve JavaScript, is to treat JavaScript as an assembly language to be targeted.

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.[108] 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.[109] Secondly, developers could write large-scale 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 type 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 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 (or some combination) 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.

---

107 http://asmjs.org/faq.html
108 http://asmjs.org/faq.html
109 http://www.youtube.com/watch?v=BV32Cs_CMqo

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 document, which represents the consensus of many JavaScript leaders at Google, including representatives from the ECMAScript 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.”[110] 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].”[111] 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).”[112] 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

110 Subject: "Future of Javascript" doc from our internal "JavaScript Summit"
111 Subject: "Future of Javascript" doc from our internal "JavaScript Summit"
112 Subject: "Future of Javascript" doc from our internal "JavaScript Summit"

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 in key to its widespread adoption. “[Dart] is designed to keep the dynamic, easy-to-get-started, no-compile nature of Javascript that has made the web platform the clear winner for hobbyist developers.”[113] JavaScript can written by amateurs without real tools or training. Google believes this aspect of the Web is important to maintain. And lastly, Google ultimately 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.”[114]

Example of 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. 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

113 Subject: "Future of Javascript" doc from our internal "JavaScript Summit"
114 Subject: "Future of Javascript" doc from our internal "JavaScript Summit"

applications in a language designed for the task.

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 and 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, we still have to run JavaScript in the browser, which is not nearly as fast as compiled languages. With TypeScript, could we write Photoshop for the Web? Or write immersive 3D games for the Web? No. At least not in the short term. TypeScript might bring us 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 we write Google Docs with Emscripten? 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 we got a second chance at 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 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 is clearly against the interests of other browser makers. Dart might be like C#. C# was written by Microsoft to try and 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. Because the platform is entirely controlled by Apple, iOS gets to move quickly and is not 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 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 remark 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.” As we move forward however, the Web will have to compete on its merits. The Web will be judged on its ability to bring real services and products that people want to use. In 2013, the future of the Web rests of the future of JavaScript. What a strange situation for Mocha to find itself in.