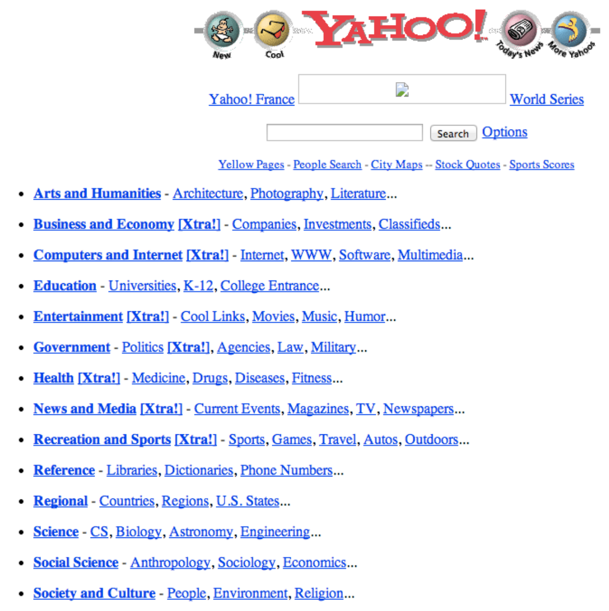
Chapter 3: Evolution and Revolution

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."[1] 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 massive 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.'"[2] Yahoo! was born.



Yahoo! France World Series

Search Options

Yellow Pages - People Search - City Maps -- Stock Quotes - Sports Scores

- **Arts and Humanities** - Architecture, Photography, Literature...
- **Business and Economy [Xtra!]** - Companies, Investments, Classifieds...
- **Computers and Internet [Xtra!]** - Internet, WWW, Software, Multimedia...
- **Education** - Universities, K-12, College Entrance...
- **Entertainment [Xtra!]** - Cool Links, Movies, Music, Humor...
- **Government** - Politics **[Xtra!]**, Agencies, Law, Military...
- **Health [Xtra!]** - Medicine, Drugs, Diseases, Fitness...
- **News and Media [Xtra!]** - Current Events, Magazines, TV, Newspapers...
- **Recreation and Sports [Xtra!]** - Sports, Games, Travel, Autos, Outdoors...
- **Reference** - Libraries, Dictionaries, Phone Numbers...
- **Regional** - Countries, Regions, U.S. States...
- **Science** - CS, Biology, Astronomy, Engineering...
- **Social Science** - Anthropology, Sociology, Economics...
- **Society and Culture** - People, Environment, Religion...

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.[3] 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 structure," Larry Page, one of these students, told Wired Magazine in 2008.[4] The Web, Page and his partner (Sergey Brin) theorized, was the largest graph ever created.[5] 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

1 http://www.theguardian.com/business/2008/feb/01/microsoft.technology
2 http://blogs.static.mentalfloss.com/blogs/archives/22707.html
3 http://visual.ly/brief-history-web-standards
4 http://www.wired.com/wired/archive/13.08/battelle.html
5 http://www.wired.com/wired/archive/13.08/battelle.html

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.[6]

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.[7] 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 scientists at research universities, the Internet's new open Web became a multifaceted platform for almost everything. Amazon proved that the Web could be used as a storefront. Online forums and blogs proved that the Web was a place for hobbies and politics. MySpace and Facebook proved that people wanted to socialize and advertise on the Web. Not only did the graph of pages and links that make up the Web grew exponentially, the cultural impact of the network grew as well. 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.

Growing Pains.

As the Web became more mainstream, users demanded more and more webpage interaction. Flashing banners, buttons that responded when clicked, 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."[8]

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. This meant that good JavaScript ideas could travel virally through the Internet. But more importantly, JavaScript

6 http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html
7 http://www.internetworldstats.com/emarketing.htm
8 http://www.w3.org/DesignIssues/Principles.html

had direct access to browser APIs, the HTML it was embedded in and the styling of that HTML. Although JavaScript was arguably much less powerful than 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 “real” 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 quickly 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. [CITATION] 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.[9] Fierce competition had pushed the two companies to 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.[10] 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.

Of course, 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.[11] Douglas Crockford, a well known JavaScript developer and somewhat of a JavaScript historian, notes that Ecma was not

9 http://www.webstandards.org/about/history/
10 https://developer.mozilla.org/en-US/docs/Web/HTML/Element/blink
11 The Past Present and Future of JavaScript (Rauschmayer) 5

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.'”[12] Netscape could not take no for an answer however. Microsoft's adoption of JScript had made it clear that the language was not just Netscape's anymore.

After trying a few more standards boards, Netscape ended up at Ecma, originally founded in 1961 as the European Computer Manufacturers Association.[13] Ecma was, as Crockford notes, “a long way to go for a California software company.”[14] 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.”[15] 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.[16]

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.”[17] 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.[18] ECMAScript 5, the more incremental release, was agreed upon in December 2009.[19] 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.[20] 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 and even the oldest browsers implement ECMAScript 3.[21]

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 boards. 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.

12 http://www.youtube.com/watch?v=t7_5-XYrkqg
13 http://www.ecma-international.org/
14 http://www.youtube.com/watch?v=t7_5-XYrkqg
15 https://mail.mozilla.org/pipermail/es-discuss/2006-October/000133.html
16 http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf
17 http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December %201999.pdf
18 The Past Present and Future of JavaScript (Rauschmayer) 10
19 http://css.dzone.com/articles/brief-history-ecmascript
20 https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions_and_function_scope/Strict_mode
21 https://developer.mozilla.org/en-US/docs/Web/JavaScript

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.[22] Although most of the components of the Web already existed (for instance the Internet, HyperText and multi-font text objects), no one at the time saw the larger potential of these components. "It was a step of generalizing, going to a higher level of abstraction, thinking about all the documentation systems out there as being possibly part of a larger imaginary documentation system," Berners-lee said in 2007 interview. He and his team created the HyperText Transfer Protocol (HTTP) and Berners-Lee himself wrote the first web client and server in 1990.[23]

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.[24] 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.[25] Amazon is another example of a successful early application on the Web.

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 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. And JavaScript, the "toy language," 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

22 http://www.achievement.org/autodoc/page/ber1int-1
23 http://internethalloffame.org/inductees/tim-berners-lee
24 http://www.w3.org/Protocols/HTTP/AsImplemented.html
25 http://www.paulgraham.com/first.html

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]."[26] 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."[27]

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 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.[28] 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."[29] 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."[30]

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 development. 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 at all).[31] JQuery, introduced in 2005, adds intuitiv Web-specific idioms, including built-in Ajax, CSS and JSON support. For example, the following code, which would 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) {
```

26 http://www.alexhopmann.com/xmlhttp.htm
27 http://www.alexhopmann.com/xmlhttp.htm
28 http://www.alexhopmann.com/xmlhttp.htm
29 http://www.aaronsw.com/weblog/ajaxhistory
30 http://www.aaronsw.com/weblog/ajaxhistory
31 http://w3techs.com/technologies/overview/javascript_library/all

```
    $("#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; by 2006, it was clear to Google leadership that the kind of horsepower they needed was not being supported in the browsers.[32]

The development of the Chrome browser, which began in 2006, was a strategic move on Google's part. The quality of the Web browsing experience was central to their business and the company wanted the browser market to innovate. Peter Kasting, a Chrome 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."[33] 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."[34]

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."[35] The JavaScript engine has been largely successful in this stated goal.[36] Mozilla, the company that was born out of Netscape's demise, 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.[37] 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.[38] Although Mozilla probably would not admit that pressure from Google was part of their decision to develop TraceMonkey, it probably did not hurt. TraceMonkey was up to 40 times faster than Firefox 3's SpiderMonkey, and it was about on par with V8.[39]

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

32 http://blogs.wsj.com/digits/2009/07/09/sun-valley-schmidt-didnt-want-to-build-chrome-initially-he-says/
33 http://www.tomshardware.com/news/google-chrome-web-browser-mozilla-firefox,14378.html
34 https://developers.google.com/v8/intro
35 http://www.youtube.com/watch?v=hWhMKalEicY
36 https://developers.google.com/v8/design
37 https://brendaneich.com/2011/06/new-javascript-engine-module-owner/
38 http://arstechnica.com/information-technology/2008/08/firefox-to-get-massive-javascript-performance-boost/
39 http://arstechnica.com/information-technology/2008/08/firefox-to-get-massive-javascript-performance-boost/

Mozilla wanted to “get people thinking about JavaScript as a more general-purpose language.”[40] 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'”[41] JavaScript is certainly not slow anymore. The language's performance has experienced an order of magnitude improvement.

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.[42] 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. Like nothing. It had zero concurrency.”[43]

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.”[44] 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.”[45]

There were a few reasons that Dahl thought that JavaScript was the perfect candidate for writing 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. 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.”[46]

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.”[47] 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.”[48] 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,

40 http://arstechnica.com/information-technology/2008/08/firefox-to-get-massive-javascript-performance-boost/
41 https://brendaneich.com/2011/06/new-javascript-engine-module-owner/
42 http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/
43 http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/
44 http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/
45 http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/
46 http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/
47 http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/
48 http://tagsoup.github.io/blog/2011/10/05/ryan-dahls-history-of-node-dot-js/

the project has had a much larger impact. Node was the first serious implementation of JavaScript outside of a browser. Therefore the choices that it has 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 to perform filesystem tasks, can easily rely on modules written by other people:

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

Therefore although it is built for writing Web servers, Node opens up the entire operating system for JavaScript programs. Node carves out a vast new territory for JavaScript, and gives it the tools it needs to get there. Not only does Dahl's work on Node free the Web server from a global lock on a single thread, it *frees JavaScript from the browser.* The popular perception of the language, and its trajectory, has been changed forever.