JavaScript How we got here. An indepth history.

Article · September 2020

CITATIONS
0

READS
1,785

1 author:

Michael Jaroya Ohuru
Jomo Kenyatta University of Agriculture and Technology
2 PUBLICATIONS **1** CITATION

SEE PROFILE

Article first appeared in: talwork.net

BY: Michael Jaroya

Introduction

"Java is to JavaScript what car is to carpet" - **Chris Heilmann**

But what exactly is a car to a carpet!? Expensive unto cheap? General purpose unto purpose specific? A piece of scrap unto a piece of rag, or should I say sh*t unto sh*t?...or maybe, Chris meant that the two are very different in design structure and/or purpose? I don't know the answer! I'm not Rick Sanchez! But assuredly I can say unto you, in the beginning, Java had a mission to be everywhere, and at its inception, JavaScript had a mission to be as popular as Java. Yea, I get it, to be as popular as Beiber is a pretty 'interesting' mission, but soon another star will shine the way for you to follow. But to be everywhere is also a 'sick' challenge per se. Java, are you matter? I mean, you are at most, an idea crafted to a formal language. Or you think that if you do occupy space, then you equal matter? Wait a minute, how much exactly do you weigh? So how is it that you want to be everywhere!? Yuck, to both of you! These two languages had intertwined their destinies unknowingly. Yet I have heard organisms solemnly swear that they hate JavaScript. That there is no beauty at all in JavaScript. Hola, naysayers, how dare you strike the hands that feed thee!? Maybe you don't understand how JavaScript is involved with your lunch at the table. Here is how.

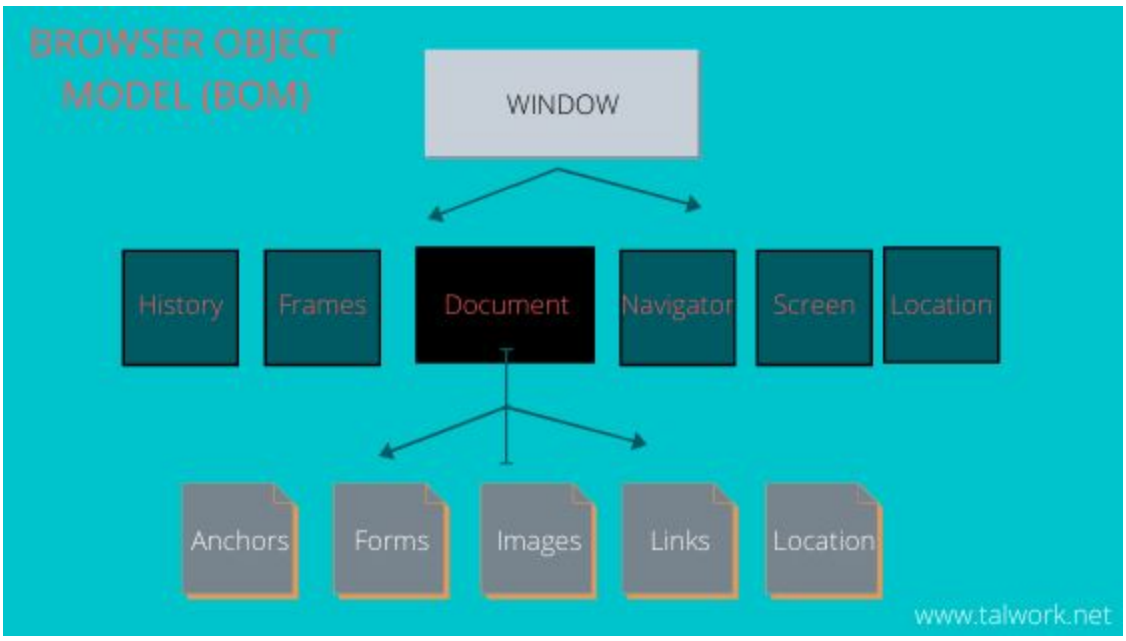
Browser Wars Begin

The Mosaic browser, Netscape Navigator, and Internet Explorer (IE)

1990, Tim Berners-Lee (TimBL) invented the world wide web (WWW), the first web browser, the first web server, and HTML. Yet by 1994, a time when Hakon Wium Lie, TimBL's workmate at CERN was proposing **Cascading Style Sheets** (CSS) for styling the web, people still didn't understand what the heck the Internet is, yet a technology that began in the early '60s. Hakon Wium Lie was also the CTO of **Opera Software** from '98 till 2016. Since TimBL's aha moment, other people were also inspired to create their own browsers so as to tap into the potentials of the web. A few projects like Erwise and ViolaWWW web browsers succeeded. ViolaWWW would later on (1993) inspire two programmers working at NCSA (at the University of Illinois at Urbana Champaign), Marc Andreessen and Eric Bina to create the **Mosaic** browser. The first web browser of its kind that displayed images inline with text instead of displaying them on separate windows. Keep in mind that JavaScript still doesn't exist in the picture yet the Java language project had already begun in 1991. Java, at its inception, was designed for interactive TV but proved too advanced for the digital cable TV industry of the time. Too advanced, you say! Hmn! You better change before change changes you. The C programming language was created in 1975 while C++ appeared 10 years later, '85.

The Mosaic browser brought WWW and the Internet to the mainstream. Andreesen later in the same year co-founded **Netscape**, the company, and then developed a new browser, **Netscape Navigator in 1994.** Within two years, Netscape Navigator dominated the browser market share at 80%. But this also meant the decline of Mosaic which was later on **licenced by Microsoft (**founded in 1975**) and its source code used to create Internet Explorer (IE) in August 1995.**

The Grand old BOM and the birth of Mocha in 10 days



All there is to the browser at this time, in a programmers point of view, is the grand old **Browser Object Model (BOM)**. This as an object, has a property called the **Window,** which in itself is also an object containing other objects as its properties like the **Navigator object, Location object, Document object (DOM), the Screen object, and History object** etc. In a sense, they are the interfaces (APIs) the browser exposed. Andreesen later (still in '95) realized that there was a need for interactivity in the browser, a way to manipulate these browser objects to have dynamic content rendering. **Java** language (developed by **James Gosling** at **Sun Microsystems**, now owned by **Oracle**) being super-trendy at the time was the favourite language of choice, but its integration into the browser seemed futile. Netscape then recruited **Brendan Eich** and tasked him to create a language that resembled Java and then add it to the browser. He ought to be done like yesterday, a clear problem faced by many software developers and engineers, that of a deadline. Well, if the line is dead then doesn't it cease to exist? If so, then what the fudge?

Mocha becomes LiveScript and later JavaScript

Eich was motivated and **10 days** later, he completed his assignment. He named the language **Mocha** and it ran on **SpiderMonkey**, a language engine (more on this in Node.js section) that he also developed in C/C++. Mocha syntactically resembled Java e.g the brackets but Eich added other beauties to the language like **first-class functions**, **dynamic typing** and **prototypal inheritance** etc. Of Course there were mistakes made in some of the decisions made during the language design (e.g the use of '==' operator) given the time constraints but Eich built a malleable/flexible language that developers could use to apply their own design patterns to. This would later on play a significant role in the Nodejs era. In September the same year (1995) Mocha was renamed **LiveScript** then shipped with the **Netscape Navigator 2**. In December the same year LiveScript was renamed **JavaScript** so that it could flow in fame with the trendy Java.

XML, ECMAScript, and AJAX

In 1996, word wide web consortium (W3C) began the development of **XML**, a language that defined the rules for encoding documents in a format that is both human and machine readable. The same year, Microsoft reverse engineered JavaScript and named their version **JScript** and shipped it with their **Internet Explorer** browser. And so existed, in the mainstream browsers, two different implementations of the same language that obviously would lead to a divide in the developers' world. Netscape joined forces with Suns Lab in order to rival the big Microsoft in the browser wars. Soon, Netscape seeked for standardization of the JavaScript language and this was denied by the mainstream standardization bodies but **ECMA**

International agreed to standardize the language. ECMA released their standardized version in June 1997 and named it ECMA-262 a.k.a **ECMAScript**. But why not name it JavaScript? Unfortunately, Suns Lab claimed the ownership rights to the name 'JavaScript' since they owned Java. So, ECMAScript and JavaScript are one and the same thing. Developers just tend to refer to ECMAScript as the standard upon which JavaScript's implementation is based. The names are to be used interchangeably. The standardization gave the browser vendors a nice specification to use while implementing JavaScript in their browsers.

The Internet Explorer version released in 1996 introduced the **iframe tag** which could fetch and load content asynchronously. In '98 Microsoft's **Outlook Web Access** team developed the concept behind **XMLHttpRequest** scripting object which expounded on the iframe tag. Also in 1998, Brendan Eich co-founded the **Mozilla Project** whose core purpose was to help manage the open-source contributions of the Netscape source code that was now open to the public. Internet Explorer version 5 was released in 1999 and it introduced the **XMLHttpRequest object** to the world which was later on implemented by browsers vendors as XMLHttpRequest JavaScript object. Basically, it's a group of technologies collectively called **Asynchronous JavaScript and XML (AJAX).** AJAX allowed for asynchronous client-server communication without full page reload, and implemented **XML** standard as the defacto language for encoding the data sent between the client and the server. AJAX tremendously improved the user interactivity in the browser.

The Dot.com Market Crash

Improved browser technologies brought about so many web applications and companies backing them. There were email apps, Internet messages apps, websites as company front-ends etc. Businesses were booming. Companies acquired companies as their internet stocks boomed. As a consequence, **Netscape was bought by AOL in 1999**. So many Internet companies inflated their values over the promises the web presented and spent recklessly on advertisements to gain tractions into their websites, and so began the impending **dot.com**

market bubble crash, that would at the beginning in the mid ‘90s, drive the progress of technology up as the bubble itself rose higher and higher. Companies like **Amazon** (founded in 1994), **eBay** (founded 1995) and **Cisco** (founded in 1984) really gained traction at this period. But as the saying goes, everything that goes up, must eventually come down. The **Nasdaq Composite stock market index** rose to 400% between 1995 to 2000 and then the bubble burst in 2000, and the index fell to 78% in 2002. To get a feel of what this means, if your internet company is initially (100%) valued at $1M, the boom would have raised it to 400%, $4M. The market fall would leave you at 78%, $780K. Online companies like **pets.com** ceased to exist while big companies like the aforementioned incurred heavy losses but soon recovered and went on to dominate the market share in their respective industries, because of the death and absence of competition. That’s how **Google (founded in 1998), Amazon, eBay and Cisco** amongst others found their domineering feet. Blame them not, for it is wise for the hawker to sell his items during extreme cold or hot sunny afternoons because his competitors are probably asleep under the blankets or resting out the hot sun.

Netscape dies and out comes Mozilla Firefox

By the year 2000 Internet Explorer’s market share rose up to 80% while that of Netscape Navigator declined and AOL was compelled to shut down the Netscape browser unit in 2003. Brendan Eich then founded Mozilla Foundation (non-profit) where he used his SpiderMonkey JavaScript engine to create **Mozilla Firefox** browser in 2003. He also founded Mozilla Corporation, a profit-making company for maintaining Mozilla Firefox browser. Eich in 2016 used the open source **Chromium** project to create the **Brave browser** that blocks ads and trackers. A year later, he co-created **Basic Attention Token (BAT)**, a cryptocurrency designed to run in Brave browser. Alongside Internet Explorer’s fame in 2000, was a mistake. The development team ended up making their own fragmentations of JavaScript language, not found in the ecma standard. Those, obviously, were not implemented by other browser vendors since they weren’t part of the standard. So if you see the below code, think of this mistake.

```
<!--[if IE 7]>
 <style type="text/css">
   div.ie7 { display:block; }
 </style>
<![endif]-->
```

The revolutionary JSON and code linting

2003 also saw the creation of **JavaScript Object Notation, JSON** by **Douglas Crockford**, an employee at **Yahoo (**founded in1994**)** who happened to be on the ECMA team as a delegate from Yahoo. He saw the bulkiness of XML and how resource intensive it is for client-server communication and then created a slick piece of technology for data representation that opened up the frontiers of so many other technologies such as document oriented databases like **MongoDB**. JSON would later on be implemented in EcmaScript 5. Crockford is also famed for creating **JSLint**, the first linter (a tool for flagging programmatic errors) for JavaScript. He is also famed to be the first person to find out that JavaScript had good parts, and ofcourse he is the author of the book, ***JavaScript: the good parts***. He also recently, in 2018, released another awesome book, ***How JavaScript works.*** He is currently working on a **better language than JavaScript.** I really look forward to it.

The great split, ECMAScript v3.1 vs v4, the birth of Adobe Flash

Greatest of all, a story is told of a sharp disagreement that erupted within the EcmaScript committee on the direction that the next version of EcmaScript (version 4) was to take. The chief antagonist was Douglas Crockford. EcmaScript 4 proposed features like **type annotations, classes, interfaces**, etc. Features that when implemented would ready JavaScript for use in building enterprise applications, or so they claimed. Crockford feared that EcmaScript 4 was too big and would soon get out of hand. Microsoft had his back and they rather proposed an incremental change to EcmaScript 3. The two warring sides split and this led to **two new EcmaScript proposals, version 3.1 by Crocford, Microsoft and company, and Version 4 by the protagonists**. The war would go on until 2008 when EcmaScript 4 was eventually scrapped. But EcmaScript 4 did leave a footprint in the world. **Adobe** (founded 1982) implemented it to build their own language called **ActionScript** which they used to drive **Flash.** The discontinued support for EcmaScript 4 also meant that Adobe would have to maintain their code base alone without any new JavaScript backings. We all know what

happened to Adobe Flash early this year (2020). " ***By the end of 2020, Flash will be gone from all supported Windows versions and users will no longer have any ability to enable or run Flash."*** I used to study from a website that had all their animations built with flash, now I have to view the source code of the pages and grab the flash file then run it in my PotPlayer, else, I learn nothing at all.

Make JavaScript easy with jQuery, drive things faster with V8 on Chrome

In the mid 2000, JavaScript was still frustrating to many developers trying to develop cross-browser applications, majorly because of its heavy syntax and the technical documentation. This inspired **John Resig** to develop **jQuery in 2006**. With a well written documentation and a way lighter syntax compared to JavaScript. jQuery enabled developers to build far more complex applications that ran more reliably on all browsers and with a lot of ease. More on jQuery can be found in Express.js section. Other frameworks like **Prototype**, **Dogo**, and **Mootools** also came up trying to do the same thing, but jQuery won. As if not enough, the mid 2000s (2008, to be specific) also saw the release of **Google Chrome** and the powerful **V8 JavaScript engine** (more on these can be found on Node.js section). V8 engine changed the way JavaScript was compiled and interpreted making it a viable option for high performance apps both on the client side and on the server side, since it could run on both sides. V8 would later on inspire many projects including Node.js whose story is told next. The JavaScript story will resume in the **VERBAl** section.

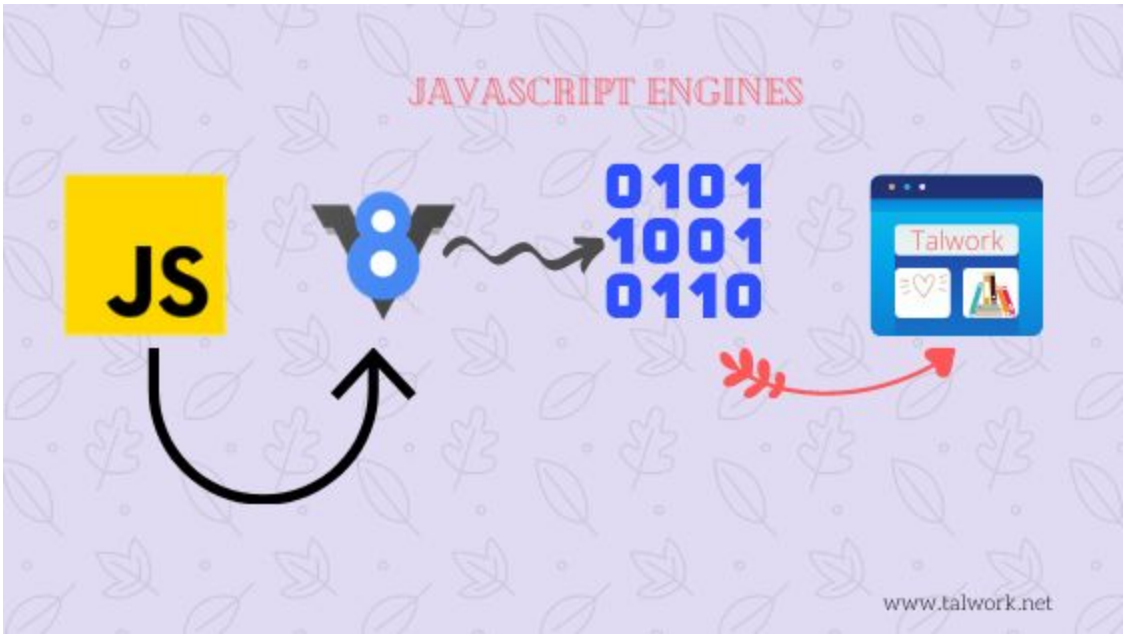
The Node.js era begins

Engine	Status	Steward	License
WebKit	Active	Apple	GNU LGPL, BSD-style
Blink	Active	Google	GNU LGPL, BSD-style
EdgeHTML	Active	Microsoft	Proprietary
Gecko	Active	Mozilla	Mozilla Public
Servo	Active	Mozilla	Mozilla Public
Goanna	Active	M. C. Straver[5]	Mozilla Public
NetSurf	Active	hobbyists[6]	GNU GPLv2
KHTML	Discontinued	KDE	GNU LGPL
Trident	Discontinued	Microsoft	Proprietary
Presto	Discontinued	Opera Software	Proprietary

Node.Js is a cross-platform JavaScript **run-time environment** that executes JavaScript code outside of a web browser. But what does that mean? To begin with, in the traditional sense (before Node.js era began, in 2009) JavaScript code could only be executed inside a web browser. But why, so you might ask? A web browser as an application (a chunk of code; mostly

C++ code), consists mainly of two pieces of sub-applications amongst others: the **browser engine** which also contains the **rendering engine**, and the **JavaScript engine**. The **browser's engine** handles the communication between the web browser's user interface and the **rendering engine** which, in turn, is responsible for rendering all those html, css, and data passed to it (web pages). Notable browser engines include: **WebKit** that runs Apple's Safari browser, **Blink** that runs Google's Chrome browser and Opera browser, **Gecko** that runs Mozilla's Firefox browser , and **EdgeHTML** that runs Microsoft's Edge browser.

How do the engines (V8, Nitro, SpiderMonkey, Chakra) work?



The **JavaScript engine** is the other sub-chunk of code. As you might have guessed, it's a program that converts JavaScript code into something the computer processor can understand. Its purpose is to take your JavaScript code as input, talk to the underlying processor to process your code, and then pass the result as data over to the rendering engine for display. Also to note, since the engine has to talk to the processor, it has to be built in a low level language like Assembly, C or C++. But how does it know how to process JavaScript? The simple answer is; were I to write my own JavaScript engine using C++, I would need a guideline/standard to follow. This guideline would define the language syntax and the expected processing procedures to produce the desired output. And yes there exists the JavaScript standard called EcmaScript. Get it here. In a nutshell, a JavaScript engine implements the EcmaScript specification. Take this piece of code as an example.

<code>let age = 47;</code>

My C++ engine will have to parse this and produce, as an output.

<code>int age = 47;</code>

The C++ compiler will then handle the output and return the result to my JavaScript engine which will then pass the results to the rendering engine. Disclaimer, writing your own engine requires a heck of manpower (clean, efficient coders), time (in years), and capital.
Since JavaScript lives and runs in the browser, it's run-time environment is the browser. Notable JavaScript engines includes: **V8 engine** (built in C++) that runs Google Chrome and Opera browsers (a rather interesting name borrowed from the powerful V8 automobile engines that run some of the world's most powerful vehicles), **SpiderMonkey** that runs Firefox (The very first JavaScript engine, created by Brendan Eich, the creator of JavaScript, and later Brave browser), **Nitro** (formerly SquirrelFish) that runs Apple's Safari, and **Chakra** that runs Microsoft's Edge and Internet Explorer (IE). But where does Node.js come in?

Dahl is frustrated with Apache, Lerdorf's machine and also PHP's engine.

Ryan Dahl, an American Software Engineer, wasn't happy with the limited capabilities of the world's most popular web server, Apache HTTP server, at handling concurrent connections (upto 10,000 and more). The sequential programming method adopted while creating the server also meant that it is a blocking Input/Output (I/O) system; i.e a processing system that can only allow one process to complete its execution first before allowing another process to commence. Then were the days. Notable shout outs go to **Rasmus Lerdorf** who wrote some, if not most, of the cgi binaries (in C language) that constituted the Apache HTTP server. Lerdorf is, of course, the creator of PHP language (no wonder the existence of PHP drives the continued existence of Apache server). He also added the LIMIT clause to SQL. Ryan Dahl, in 2009 while working at Joyent, took V8 code (written in C++, and since it's an open-source project) and created magic with it.

Dahl's Resolve and the beginning of the Better-Apache project

You must understand this; V8 as an open-source project, avails its code for everyone to see, download, use and refactor as they see fit. Importantly, V8 code can be run standalone on any OS architecture as a C++ software that interprets JavaScript, or it can be embedded into any C++ application of your own. So your application, at the end of it all, other than doing what you wanted it to do e.g manipulate the file system if you were building a file explorer, it can also interpret JavaScript code. Ryan Dahl wanted to create a new web server that solved the problems he saw in Apache. He looked at the rich and unexplored features of JavaScript and then decided to use it to create the solution. But JavaScript language is just too high level (there exists a level of abstraction between it and the OS) to enable it to handle some of the low level functionalities a web server might be required to do. These functionalities may include **ways to deal with files e.g create-read-update-delete files**, **handle zipped files and image**

files, ways to **communicate with databases**, ability to **communicate over the internet** (which is not in-built in JavaScript), ability to **understand HTTP and handle its requests and responses in the standard format** e.t.c. So what did Dahl do?

How he did it

JavaScript language sits on an abstraction layer on top of the JavaScript engines which are written in low level languages like C++ in case of V8 JavaScript engine. C/ C++ has access to Operating System's file API, the Sockets API is built right into it to enable it communicate over the internet and parse HTTP, database storage engines e.g InnoDB that runs MySQL database systems is written in C which shares the programming style and level with C++, NoSQL database drivers e.g MongoDB is written in C++ and they both expose their APIs for consumption. On top of all this, V8 is the fastest and most performant JavaScript engine out there. To make things more interesting, a C project called libuv already existed in development that provided support for asynchronous I/O based on event loops with rich features for networking (TCP,UDP,DNS) , thread handling, the file events etc. Obviously, this is what he needed to overcome the blocking nature of Apache. What more could Dahl need to be convinced to use C/C++, libuv, and V8? So he began a C++ project, let's call it Better-Apache. He imported V8 and libuv projects into Better-Apache as development dependencies. Other projects he imported include: http-parser, npm, openssl, and Zlib projects etc. He added his own C++ modules e.g crypto, buffer amongst many others. He wrapped the C++ modules above and bound their running processes through V8 to JavaScript libraries which he built. Of course

the JavaScript libraries I talk of are just like API endpoints that contain routines which when called, invoke their C/C++ implementations.

Better-Apache released as Node.js, JavaScript everywhere begins

To cut the long story short, Dahl basically took a bare JavaScript engine and added server functionalities into it. It goes without saying that Better-Apache was released in 2009 as Node.JS, a JavaScript run-time environment with web server capabilities also event-driven asynchronous. A web server that can run JavaScript. In addition to these, Node.js can run in whatever environment it is installed in. This may be your computer (not in your browser) or on a server (a computer with resources and is available, mostly via Internet, to serve the resources to a client whenever needed). When installed on the client side (your computer), other than giving you local **server capabilities**, Node.js will also enable you to write command line tools with JavaScript. When installed on a server or when running as a server on your local machine, Node.js gives you the ability to run JavaScript scripts on the server (server-side scripting) to produce dynamic web page content that will then be sent to the client (browser) for rendering. But I thought the browser also has a JavaScript engine of Its own? Sure. So does this mean that I can build an application purely on JavaScript both as the front-end language and as the back-end(server side) language? Yes It does. JavaScript lives everywhere now and Node.js provides the best solution for building real-time web applications that scale.

Simplify Node.js's web capabilities with Express.js

As jQuery is to JavaScript so is Express.js to Node.js. JavaScript is a complete language on its own, it can do anything including writing dynamic web pages, embedded systems programming, AI and machine learning etc. Atwood's law states that, "**any application that can be written in JavaScript will eventually be written in JavaScript.**" Amongst the many things JavaScript can do is its ability to manipulate the DOM. But doing this in pure JavaScript is tedious.

**<code>/*
Using plain vanilla JavaScript, selecting an element and changing its properties */
document.getElementsByClassName("myclass").style.color = "red";
//Or using modern DOM traversal, you may write
document.querySelectorAll(".myclass").style.color = red;</code>**

But with jQuery, you will write a concise line of code like
<code>//Jquery select and modify color of an element

$(".myclass").css("color", "red");</code>

So, in a nutshell, jQuery is a chunk of code (library) written in JavaScript whose purpose is to simplify HTML DOM tree traversal and manipulation, as well as event handling, CSS animations, and AJAX.

Node.js also can do many things; but amongst those things is its role of creating web applications (majorly handling the HTTP and FTP protocols). Handling HTTP requests and serving files in pure Node.js is also tedious. A simple server instance in Node.js would look like:

```
<code>
var http = require('http');
//create a server object:
http.createServer((req, res)=> {
 res.write('Hello World!'); //write a response to the client
 res.end(); //end the response
}).listen(3000,()=>{
  console.log("Server listening on port 3000");
}); </code>
```

But with Express.js you may write;

```
<code>var express = require('express');
var app = express();

app.get('/', (req, res)=> {
 res.send('Hello World!');
}).listen(3000,()=> {
 console.log('Server is listening at 3000')
})</code>
```

Which already includes route handling other than being simple to understand. Of course, the whole is way greater than the sum of parts, so don't doubt it.

The beauty of Express.js

Created by TJ Hollowaychuck in 2010 (just a year after Node.js was released), Express.js is a chunk of code (library) written in Node.js whose purpose is to simplify the process of building web applications and APIs using Node.js. It provides you with the complete toolset you need for web applications development. However, since Express.js provides generic functionalities that can be customized by additional user-written code to fit their needs, it fits the definition of a

software framework. Specifically, a web framework. In addition to these, Express.js is minimal in footprint and also, will not abstract the Node.js code you are already familiar with (it will only simplify what's difficult to do with Node.js, leaving the rest for Node.js) and is un-opinionated (lets the user chose their own design methods to employ). The definition of the adjective, **express**, should speak for itself; it means: **clearly indicated; distinctly stated; definite; explicit; plain**.

A need to be more VERBAl at the front-end, Vue, Ember, React, Backbone,and Angular (VERBAl)

" Aigo! The servers have been doing all the talking for a long time now! And what have we been doing in return? Just listening! Yea, they pass to us completely rendered pages and then order us to display them. And like sheep unto the shepherd, we simply obey. Enough of this! We also want a taste of the raw data! We can also talk, for Pete's sake! I, Emperor Constantine of the Client-Side Empire, therefore decree that, in this vulnerable year of 2010…"

Web 1.0, 2.0, and Server-Side Rendering

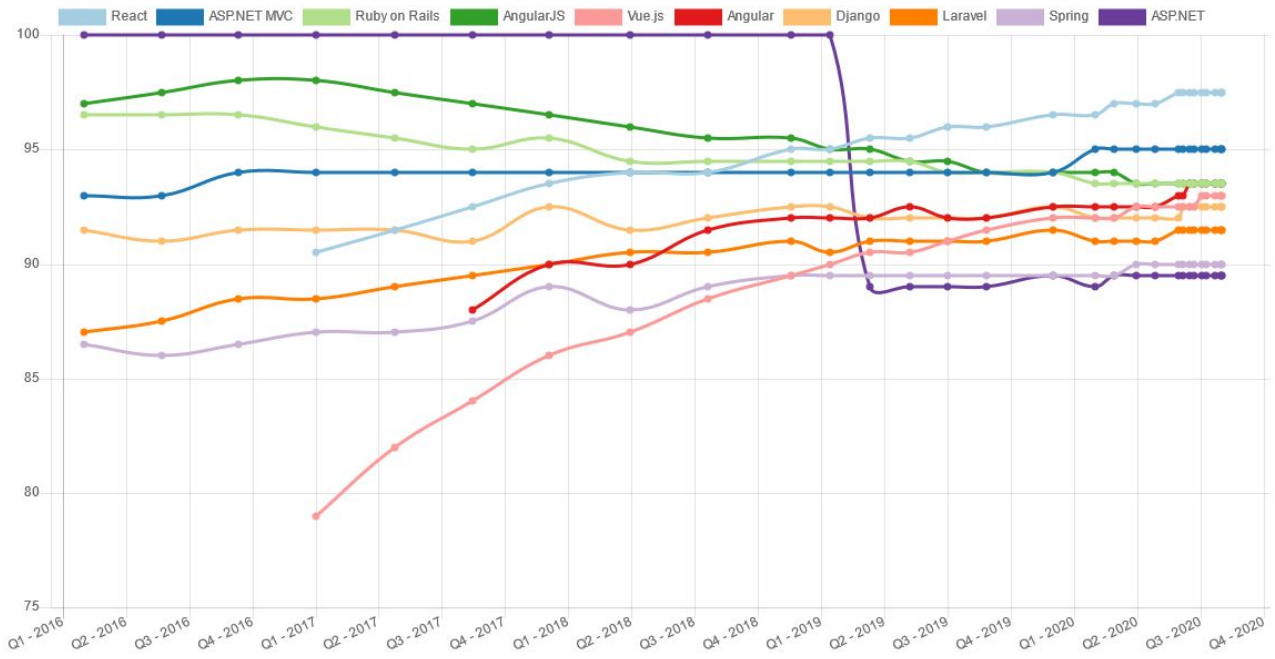
Stay with me, folks. On rewinding back to 1996, JavaScript and CSS were at their infancy. The type of web pages presented on Internet Explorer and Netscape were pretty much static. To mean, once a web page is displayed, it stays the same lest another page is requested from the server and re-rendered then displayed. This was the Web 1.0 era. But as time passed by, JavaScript matured and in the early 2000s, the browsers were displaying dynamic HTML. The browser objects (window, screen, document, history, navigation, history) could now be manipulated by JavaScript as below.

<code>window.innerHeight //Prints the height of the browser window
window.innerWidth //prints the width of the browser window

screen.height // Prints the height of the user's computer screen
screen.width //Prints the width of the user's computer screen

history.back() //navigates to the previously visited page
history.forward() //navigates to the next page
/*
play around with the rest of the objects and their properties destructured below
***/**
let {href, hostname, protocol} = location; //e.g location.href
let {appName, appVersion, cookieEnabled, online} = navigator;//e.g navigator.online
let {cookie, getElementById, querySelector } = document;//e.g document.cookie
</code>

Other features like drag-n-drop and CSS animations were also introduced. As I have told previously, this led to the rise of web applications like wikipedia, Amazon, YouTube. This era is called Web 2.0. To note however, the business logic of all these web pages and apps were done on the server. For example, the decision of what content to display (and how to display it) when a user selects A or B would have to be determined by the server, more specifically, the server (backend) language e.g PHP. This proved very expensive (time being the commodity here), as the user would have to wait for the server to make the decisions, render the content and then return it before the client displays it. Only then will the user see the result. A simple task like error checking on a form would also follow this procedure. AJAX (released in 2006) abstracted these client server handshakes from the user as there were no needs of refreshing the whole page just in order to reload a sub-content. These would happen on the background and when the content was returned to the client by the server, the client would only reload the necessary section(s).



Client-Side rendering and the dawn of Front-End frameworks

All this time, heavy machinery (JavaScript engines) are just lying idle in the browsers. I remember, with nostalgia, creating all my web pages and doing all the logic with PHP or Python and never touching JavaScript (that was as late as 2017). To be sincere, I tried my level best to avoid the language because I often wondered why I needed both JavaScript and PHP. I would assign mundane tasks like image scrolling to JavaScript and maybe sometimes DOM manipulation but the rest I did in a server language. Well, I was a novice developer.
Nota bene: **A framework is basically a library that brings organization into an application by creating reusable code. When used, a framework provides a common ground for all developers to understand one another. It kind of standardizes how things are done and this brings about communities around them resulting in collaborations.**

BackboneJS, AngularJS, EmberJS, Bootstrap, CoffeeScript and transpiling.

Then there was an awakening in the JavaScript world in 2009. A cleaned version of ECMAScript, version 5, was released. It was built upon version 3.1 after version 4 was completely scrapped, Crockford added that, “having 10 years in which nothing happened allowed for the language to become stable.” A year later (2010), developers tried to leverage the power of the sleeping engines and this saw the release of JavaScript frameworks that specifically targeted single-page applications with their business logic on rendering done on the client side. It also

saw the rise of CSS frameworks. **Backbone.js** and **AngularJS** were first to be released in 2010. Both tried to manipulate DOM based on the business logic but they did it differently. Backbone decided to go lightweight and chose imperative programming while Angular decided to be all inclusive and chose declarative programming style. AngularJS, being all inclusive, ended up being heavy and developers were faced with a deep learning curve. Backbone.js did not follow the typical Model-View-Controller (MVC) design pattern that was gaining traction then. Bear in mind that MVC was introduced way back in 1970 in **Smalltalk** programming language. Backbone.js did not implement Controllers. AngularJS on the other hand, did not implement models leaving you wondering where the hell you're gonna stone your data. AngularJS also introduced two-way data binding. A year later (2011), **Ember.js** JavaScript framework and **Bootstrap** (from Twitter) CSS framework were introduced. Ember.js actually implemented the typical MVC design pattern in full. CSS frameworks like bootstrap don't really care about where the page is rendered, they only care about the final page which they then style.

TypeScript, ECMAScript 6, Babel and Webpack

Jeremy Ashkenas, the creator of Backbone is also the creator of **CoffeeScript (**in 2009) and **Underscore JS**. **CoffeeScript** is "a little language that compiles back into JavaScript." Although it looked unnecessary to invent such a thing, it introduced the concept of compiling a later/higher version of JavaScript into an earlier version. This is called trans-compiling or rather, **transpiling** and would later prove very important after the release of ECMAScript 6 or ECMAScript 2015. ECMAScript 2015 introduced so many beautiful features like **promises, arrow functions, spread and rest, destructuring** etc. However, not all browsers had implemented it hence the need for transpiling back to the previous supported versions. This saw the rise of tools like **Babel (**in 2016). Transpiling also charged the development of **TypeScript** by Microsoft in 2012 which is a strict superset of JavaScript that adds static typing to the language. Its code however, transpiles back to JavaScript before execution. JavaScript frameworks were becoming complex and heavy and this prompted the release of **Webpack** to help bundle the dependencies of these feature-rich frameworks in 2012.

The War of Components, React and Vue

Flashforward to 2013, **React (**from Facebook**)** was announced at JS Conf. It adopted Angular's declarative programming style of UI development but it was much better. It improved declarative UI with **unidirectional data flow** and **immutability**. React also introduced the concept of **virtual DOM** and applied it to render pages like Server-Side rendering does. React also introduced the concept of **components as a way of maintaining the state of an application**. A component is a little piece of reusable code that represents an item in the web page. An example may be a chunk of code that represents a list item or a sweet alert or a

navbar. This list item component can be reused as many times as possible in your application. But any change made to its data controller would reflect in all the instances of the component. This meant that code became more predictable. Components would become the de facto way of organizing React applications.

But then, "**I figured, what if I could just extract the part that I really liked about Angular and build something really lightweight,**" said **Evan You**, the creator of **Vue.js in 2014**. Evan, then a Google employee, had worked with Angular in a lot of projects and indeed felt the heavyweight punches from the framework. He seeked to use the better parts and build something cooler. But React also was in existence, so Evan chose to amalgamate the good parts from both Angular and React to create Vue.js. Instead of starting at components as the bare minimum as React does, Evan made Vue to be incrementally adaptable such that you could just import a 33kb file into your project (or use a CDN) and then focus on very easy declarative rendering to build your complete app without necessarily going for component composition which it also supports. If your project is small, then you can just go with declarative rendering else, you can combine it with components. React supports cross-platform mobile development using React-Native. Vue also introduced Vue-Native, built on top of react native, that does the same thing. A luta continua.

Now and then...

JavaScript has stabilized pretty well and now it’s the most important programming language every developer should know. The current version is ECMAScript 2019 or ECMAScript 10. JavaScript has also found its way into embedded systems, IoT devices, mobile apps, desktop apps and the list goes on and on. To watch is **WebAssembly,** a group of technologies that allows you to write Assembly, C, C++ code and then convert them to binary JavaScript and run them in the browser. Do you know how powerful and future rich these low languages are? Well, everything they can do will now be extended to JavaScript. Also to watch is **WebSockets.** I am pretty convinced that WebSockets is a faster and better way of obtaining data from an application than an API. but just keep your heads up. What is today was tomorrow; a clear proof that history is created day by day. Since this article is a history lesson, I am obliged not to conclude it for more will come.