Telerik® A PROGRESS COMPANY

THE FUTURE OF JAVASCRIPT—2016 AND BEYOND

Interviews with Developers Building the JavaScript of Tomorrow

CONTENTS

ECMASCRIPT 2016 (ES7)

If you have had anything to do with software development (even non-web dev) in recent years, you are likely familiar with the evolution of JavaScript and the fact that it is eating the world. I guess I'm not saying anything surprising here. In fact, I assume most readers have already heard, ad nauseam, the historical JavaScript details about the creation, life span and success of JavaScript.

In this article, I am not going talk about what has already happened or how we got to where we are today. Hooray, right! This is old news. Instead, I'm going to talk about the future and make a few predictions about what might happen with the JavaScript language, and the community, as we look forward into 2016.

In order to make a reasonable prediction about the future, I polled the JavaScript community with a couple of questions to springboard this article with more than just my own personal thoughts. What follows is a summary of the questions asked with a mixture of answers, containing both my thoughts and the thoughts of those who answered my questions.

Many thanks to those who took the time to answer my questionnaire about the future of JavaScript. My thoughts were challenged and enlightened on several topics.
You are: @aortiz, rodneyrehm, @softwarefloyd, @sergiopereira, Kitson Kelly, Zackargyle, bahmutov, runspired, Nicholas C. Zakas, getify, js_dev, hemanth, Brendan, Alex, Chernov, @rlsix, @briankardell, @codekult, @alexbrbr, @dfkaye, @christosmatskas, @dfernandeza, github:spaced, @assaf, @BrianDukes

Which feature added in 2015 will be considered the most critical in 2016?

A native module system may be new to JavaScript, but it is neither flashy or new to programming languages. Modules, dependency management and loading are, in fact, what most developers would consider a basic requirement. In 2016, I believe modules will be deemed the most important feature added to the language. Of course, due to its lack of native support, it might take all of 2016 for developers to actually realize it. Most of the participants in my questionnaire agreed that modules were the most critical addition, but a few did not. Promises right behind modules were considered a vital addition as well. According to @codekult, promises are the most vital addition to the language.

> "Promises, because it is the base of the perfect pairing for several other updates: generators, async functions, observables, fetch, service worker and so on."
>
> **@codekult**

Modules might not blow your hair back like a concise usage of an arrow function or the simplicity of a returned promise, but the addition of native modules is like adding that fourth leg to a three-legged chair that many non-Node developers have been sitting on in a strained and unbalanced position.

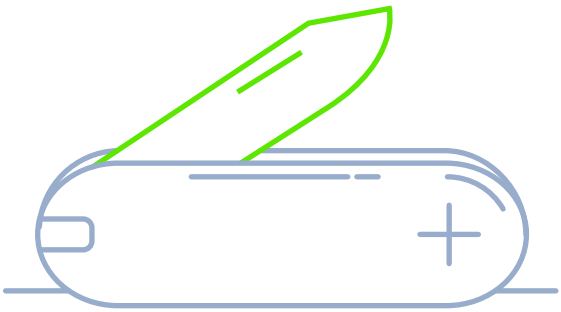
Learning the new module syntax will be easy, given that the concepts are similar to CommonJS. However, making use of JavaScript modules in production is a bit more complicated. To use ES 2015 module syntax, you'll have to use a transpiler (e.g. babel) or a tool that uses a transpiler (e.g. webpack or systemJS).

If you think about it, the addition of a native module system brings the browser in line with Node and the possibility of using a single, **native**, module syntax on both the client and the server. I'm hoping to see this materialize to some degree in 2016 so we can stop pretending with things like CommonJS and browserify. It would seem browser vendors are even in a holding pattern with things like HTML imports until ES 2015 modules are flushed through the developer communities.

One day, in the future, the native JavaScript module syntax will work everywhere and the non-native module formats (e.g. commonJS, AMD, UMD) along with the non-native loaders (browserify, webpack, systemJS) will be an unnecessary complexity from the past. In 2016, this will start to materialize, but the ideal is still many years away from being fully realized. Thus, in 2016, solutions like webpack and systemJS will still be the stopgap used to deliver the ideal of tomorrow.

If developers haven't used the ES 2015 module syntax yet, I believe they will most likely adopt it in 2016. This is mainly because the syntax is being adopted by several popular front-end tools (e.g. Angular 2 and Aurelia). I believe this reality will have a massive trickle-down effect.

What will be the most used feature in 2016, from ES 2015?



Of all the features added to the language in 2015, which one will become the most used in 2016? I've been asking developers this question for over a year and there does not seem to be a mass consensus. Some say promises. Others say generators. Many assert a pet syntactical feature.

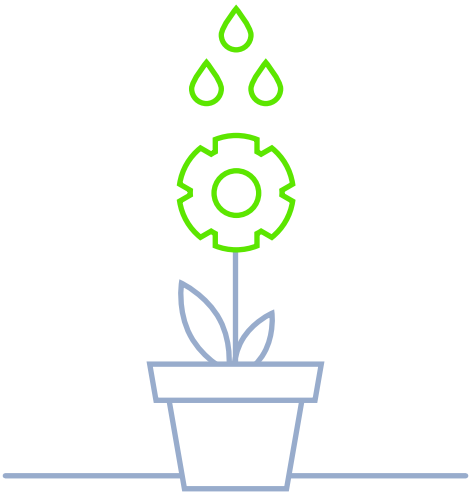
I predict that the most used feature from the update in 2015 used in 2016 will be the module syntax, and close behind that will be promises. This would make sense if, in fact, the module syntax is the most important feature, as I just asserted, thus moving forward making it the most used as well.

Of course, some people think modules could have been left out entirely. In 2016, we'll see a lot of developers holding on very tightly to the CommonJS ways of doing things.

> "Modules could have been left out for something simpler like CommonJS."
>
> **@assaf**

But, eventually, I believe most developers will accept JavaScript modules and move past CommonJS.

Will JS to JS transpilers thrive in 2016?



JS to JS transpilers are here to stay and it seems that almost everyone from the questionnaire unanimously agreed that Babel is the best JavaScript-to-JavaScript transpiler. In fact, some, like @softwarefloyd were even fearful of anything that did more than Babel.

> "I like transpilers to a point. I think ES6 and ES7 transpilers are great, but I get nervous when the type system is significantly different between the source language and the transpiled target language. TypeScript, for instance, really seems icky to me."
>
> **@softwarefloyd**

In 2016, I think you'll see more and more developers turning to transpiler source code for the following reasons:

1. A build step is more than likely already in place and adding in a transpilation step is a trivial addition in return for usage of newer ES 2015 features.

2. New tools are making use of ES 2015 features (e.g. Angular 2, Aurelia, etc...) and to use these tools you'll likely need to accept a transpilation step.

3. Web developers will want to make use of the new features introduced in the language for browser code. To do this in 2016 they will have to transpiler ES 2015 source code to ES 5 production code. (Keep in mind that Node developers using 5.x+ get a lot of ES 2015 features without transpiling.)

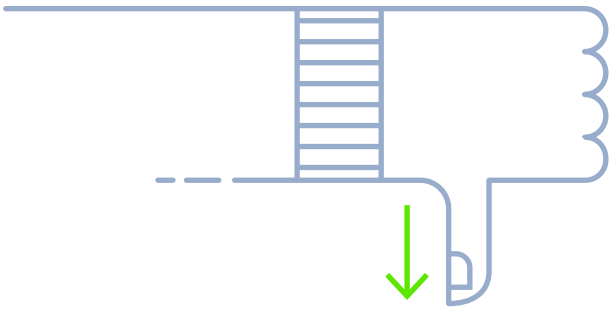
There does seem to be a small faction of developers who don't want to transpile code and would prefer to wait for proper support, believing that ES5 is enough. In 2016, the nudging from tools using ES 2015 and the usage of ES 2015 features in npm modules should negate some of this thinking.

Will run times fully support ES2015 by the end of 2016?

Realistically complete ES 2015 support could take between two and three years given the module loading hurdle. However, modern browsers and Node might just reach 90% to 95% support by the end of 2016 given the rate of change in 2015. Most of the people in the questionnaire had a similar opinion. I believe this is due to the faster than normal rate of change provided by modern browsers and transpilers supporting proposed experimental features before they land.

The thing to note here is that the most important feature from ES 2015, module syntax, will likely be the feature that takes years to become widely supported due to the loader. Of course, this is the exact reason that large numbers of developers will be turning to a transpiler in 2016 using a loader polyfill.

In 2016, which feature from ES 2015 will developers wish hadn't been added?



Given the gap between ES5 and ES6, it would seem the developer community generally believes that the new additions to the language were mostly all (badly) needed. The only exception here that seems contentious and I believe will remain contentious into 2016, is the class syntax.

JavaScript does not have classes. And, it doesn't need classes. You knew that, right. However, class syntax was added in 2015 and its just syntactical sugar over the top of the prototype-based inheritance that has always been there.

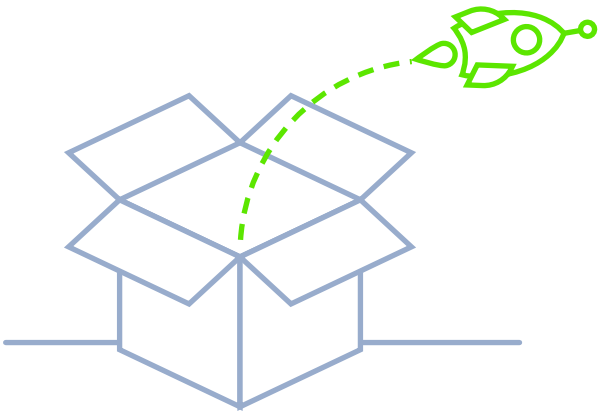
In other words, class syntax is mostly just short cut code. Its inclusion and purpose is to provide a simpler and clearer syntax for creating objects and dealing with their inheritance for those who want it. So, what's the big deal? Take it or leave it, right? The benefits to those who come to JavaScript from a traditional OOP language are obvious. So, why is class contentious?

A kind of tug-of-war is at work here. On one side, you have those who think classes are good and on the other side, you have those who think classes are evil. Will this tug-of-war be settled in 2016? Absolutely not. Why? Because not everyone approaches the design of a program with the same principles. In 2016, those who don't think like an "OOP" developer will continue to think this way. Those who do will continue to think in terms of strict encapsulation and internal state. And, well, everything will be ok. Good software will still result from both camps.

In 2016, grumbles about the addition of class syntax will continue and these anti-class developers will continue to lobby that it shouldn't be used. But, if this is the most contentious part of the massive update in 2015, I think the outlook for 2016 is good. Mostly because I think this points to the fact that those developing the language crafted an almost perfect update. The addition of the class syntax is a principled disagreement that is larger than JavaScript itself. If this is the only issue in 2016 with the update from 2015, then we are in a really good spot.

And consider that even if one can't stand its addition to the language, I would hope the gateway drug value of adding it is obvious. In other words, by providing class syntax, the JS hook can be set so those who abstain from classes will have a captive audience (i.e. actual users) in order to mount a case against them. But, again, the addition of it could just lead to a mass migration to JavaScript in 2016. In fact, I predict we'll see the usage of JavaScript continue to rise in part to additions to the language like class syntax. Once JavaScript is adopted, the normal debates of course will continue.

What will be the most important proposal finalized in 2016 and released in ES 2017?



Well, if you haven't heard, it won't be Observables. Apparently, that proposal cooked too long and burnt itself out of existence. To date the proposals likely to be added to the language next are:

- Exponentiation Operator
- Array.prototype.includes
- SIMD.JS - SIMD APIs + polyfill
- Async Functions

The first thing you should notice is that ES 2016 will not bring the same scale of changes that 2015 did. In fact, from 2016 onward, TC39 is only planning on evolving the language yearly with slight changes.

Before the news that the **`Object.observe`** proposal would be withdrawn, many believed that this addition would be the most important. In 2016, I imagine this will continue to be debated to a degree. With **`Object.observe`** out of the running, the next critical proposal is the Async Functions.

Most of 2016 for many developers will be spent learning the additions made in 2015 and how not to get "ninja stupid" with so many changes to the language. If any time is left in the coming year, I predict that most people will be learning and using async functions and promises.

Will one JavaScript package manager rise, destroying all others?

A topic of discussion in 2015 that will bleed over into 2016 (and likely into 2017) will be that of a single package manager for JavaScript. In some ways, this is a debate over the viability of a single package manager that could service both Node developers and front-end developers. Many believe that everyone should just use npm and abandon things like Bower. I find the perspective very narrow and lacking concern for a large group of website developers who don't build complex applications.

I think a single package manager servicing both the Node developer and front-end developer, if even possible, would first have to settle on a single module format for JavaScript. I'm not convinced a single package manager is ideal, but if it were there would have to be an agreement on the syntax used to construct the modules contained in the packages. This is why many have jumped on the commonJS and browserify bandwagon. Of course, the native syntax is the only logical path forward. I believe the quicker we can burn the commonJS wagon down, the faster we can get on to the native solution.

Additionally, a single package manager would have to treat both the front-end developer and the Node developer as first class citizens. Is that even possible? By combining the needs of both into a single tool, do we not risk causing more confusion and problems than fragmentation itself?

The questions I am raising will continue to be asked and answered in different ways for much of 2016 and likely remain unsolved until native module syntax and loading is natively implemented.

So, the question that is left is this: what should one do in 2016 while this is all in flux?

Personally, I think the path forward is to use something like systemJS and jspm.io. Why? Because systemJS is biased towards and tracking with the native module syntax and loading progress. Layer on jspm.io and you have a stopgap package manager that will allow the loading of packages from npm, GitHub and even bower (with plugin). I favor jspm.io because I don't think we know what the future holds for JavaScript package managers and jspm.io can literally sit on top of this issue and be future friendly to whatever is to come. So will npm win or will bower win? I say, "Who cares?" - just use jspm.io and pull from both if you need too. This alleviates some of the duplication that developers have suggested convinced them to only use npm.

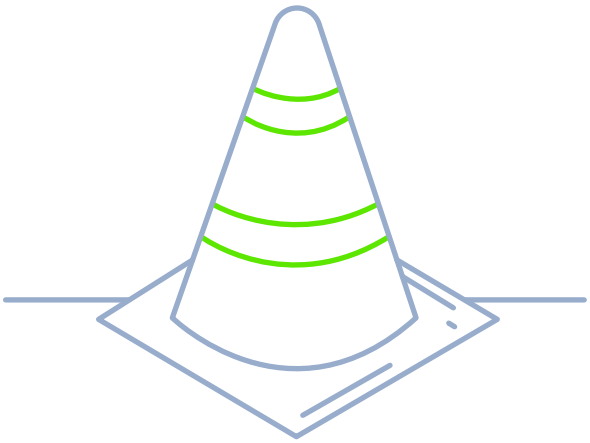
However, from what I can determine, my opinion on the matter is in the minority. A lot of people agree with the sentiment from @runspired:

> "Yes. Npm has won. It's not perfect, but there's a path to get it there, and we should work on improving npm instead of starting over."
>
> **@runspired**

A lot of developers are flocking to webpack and militantly shaming developers into using npm alone. Both webpack and systemJS solve the same core problem. I'd suggest you stay clear of dogma and simply pick whichever one makes the most sense to you and then use it to write ES 2015 modules. As for using npm alone, I say do what works best for you but don't dogmatically push a subjective mantra of npm alone onto others. Many developers don't even need a package manager, they just need a simple tool to install third party tools/code/plugins.

I've found that between systemJS & jsmp.io, there is nothing lacking in terms (except it's a stopgap) of module syntax, packaging and loading. However, some will argue that webpack does more. And it does, in fact, do more. And, if you want more, use it. Personally, if I need another tool to do another job, like serving and reloading my development code, I'll go get a focused tool that is not tightly coupled to my loader/ dependency manager.

What issues will remain in 2016 that future updates to JavaScript will need to resolve?



It's clear that even with the major update in 2015, 2016 will continue to be a year where we fill gaps in the language. In non-critical order, the unresolved topics below seem to be the pressing issues in 2016:

- How native modules are loaded in a browser will need to be ironed out and an initial implementation will need to commence.
- We haven't fully scratched the async itch. While, await functions will help, the journey is far from complete. Promises and eventually streams will need to be used throughout (e.g. HTTP promises). And O'yeah, canceling a promise. That might be a good idea.
- Concurrency and parallelism (i.e. parallel processing) in JavaScript will need to be addressed and webworkers will have to step up or step aside.
- The should we or shouldn't we debate about immutable native objects will hopefully conclude.
- Lastly, payoff whomever it takes for all browser manufacturers to treat the JavaScript runtimes in a mobile browser with the same status as a regular browser.

As you can see, we have more work to get done in 2016.

The last thing I'd like to note is that in 2016 we might also start to experience feature overload resulting in significantly different styles used for constructing JavaScript applications. Thus, in 2016 we'll need to update our thinking and education on several variations of styles that facilitate best practices.

I'm not alone in this observation, Brendan shares the same concern:

> "My concern is that there are now even more ways of doing things, so it is not as easy to squint at some code and know the layout and style."
>
> **Brendan**

With more ways of doing things right, it won't be as easy for JavaScript developers to jump from project to project. In 2016, we'll have to address this potential issue.

Will JavaScript continue to rise in use?

I don't have and I can't find a persuasive reason or opinion that would lead me to believe that JavaScript's popularity and usage will dwindle in 2016. It would seem that in terms of the immediate future, the consensus from the questionnaire concludes that JavaScript will remain on center stage in the spotlight in 2016. Beyond 2016, some, like Nicholas C. Zakas, were willing to forecast a potential decline.

> "I think WebAssembly, once available in all browsers, will start freeing people up to think about alternatives to JavaScript. The ability to compile down to WebAssembly and deliver that to the browser means we'll see people experimenting with Python, Java, Ruby and more, being written directly for the browser. Once that happens, all bets are off on the future of JavaScript."
>
> **Nicholas C. Zakas**

The reality is most people believe JavaScript will continue its dominating march in 2016 to becoming the most used programming language in the world. I should, however, mention that as Zakas mentions, there are whispers that web assembly could potentially cause a major disruption. Keep an eye on this!

Most believe, as I do, that in 2016 JavaScript will further its ubiquitous seeding by replacing more and more languages that in the past have been used to create native applications. My prediction is developers will turn to solutions like NativeScript, React Native and Electron to create native applications precisely because they want to write JavaScript alone.

JavaScript is the language of the web. What if in 2016 JavaScript became the language of native applications? If you find that to be an impossibility, I'd suggest you start wrapping your head around things like NativeScript.

FRAMEWORKS

JavaScript developers stand at the edge of a great divide. On one side are legacy browsers, differing standards, competing module systems and ES5, a language woefully inadequate for building modern applications. On the other side are evergreen standards, compliant browsers and ES6, a monumental leap forward that brings JavaScript into the age of legitimate application development languages.

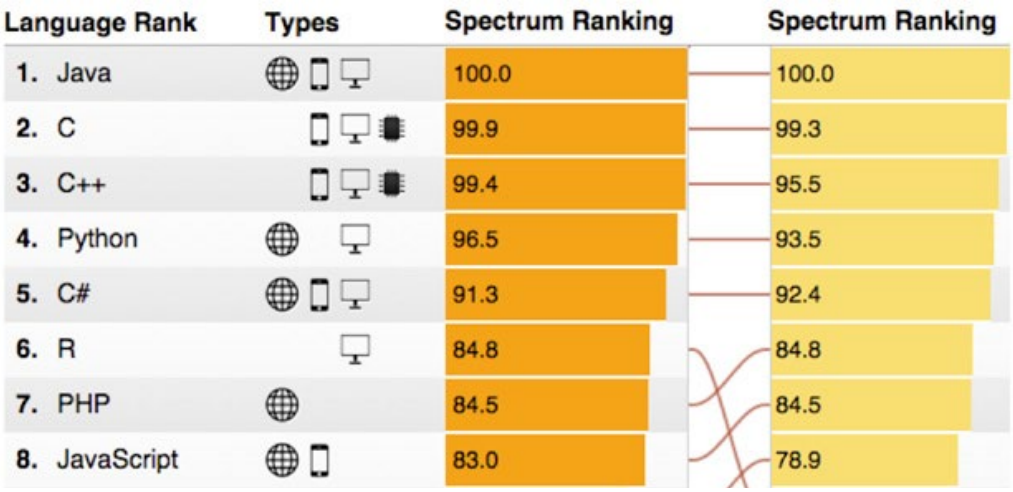
Many developers are already beginning to cross this chasm. The bridge on which they migrate is composed of JavaScript libraries and frameworks.

Libraries and Frameworks

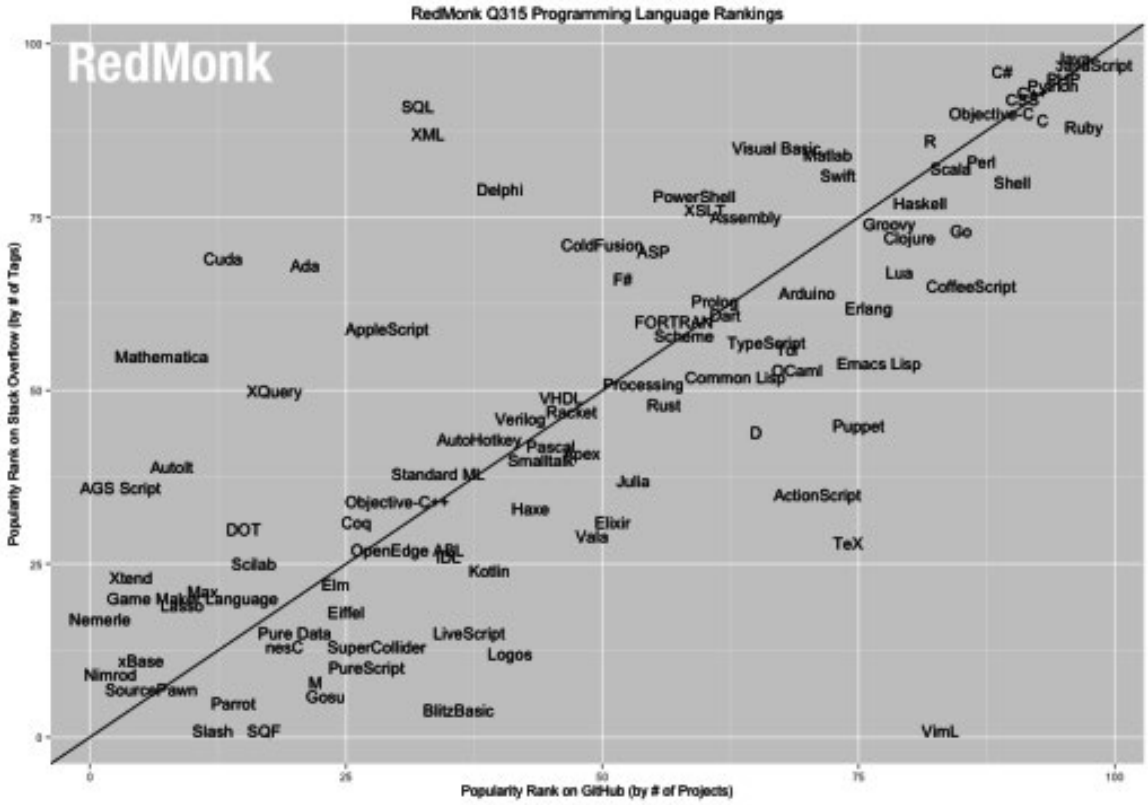
Web developers have long relied on libraries and frameworks to supplement APIs and functionality that browsers either don't provide, or don't implement consistently. The recent rise in popularity of the JavaScript language has resulted in what is commonly referred to as a "Cambrian Explosion" of JavaScript libraries, frameworks and miscellaneous tools.

There has been some debate over the true popularity of this Cambrian Explosion in JavaScript. Is JavaScript really as popular as it appears to be? It turns out this is rather difficult to measure.

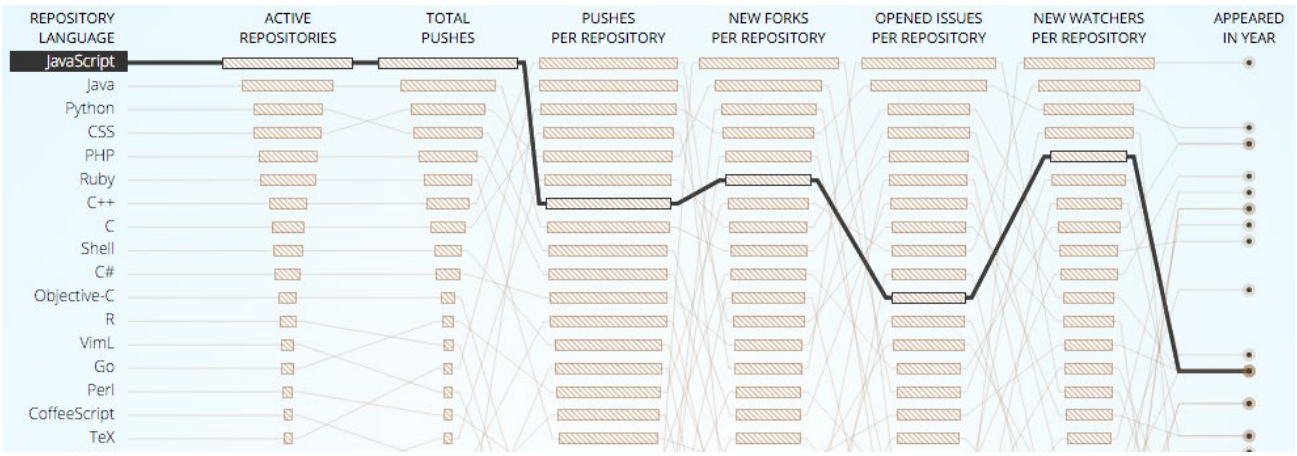
IEEE Spectrum ranks Java as the most popular language, with C as a close second. This measure is a combination of GitHub stats, CareerBuilder postings and their own IEEE Xplore library. This measurement appears to be more of a "market value" of language proficiency.



Redmonk, which measures languages relative to one another on GitHub and StackOverflow, ranks JavaScript just under Java as of June 2015.



GitHut, a site that ranks over 2 million active repositories on GitHub, ranks JavaScript as the language with the most active repositories and total pushes. Detection for languages in repos is done by the GitHub Linquist library.



It's this last GitHub statistic that pulls the curtain on the state of JavaScript frameworks and libraries. The situation is critical mass.

Critical Mass

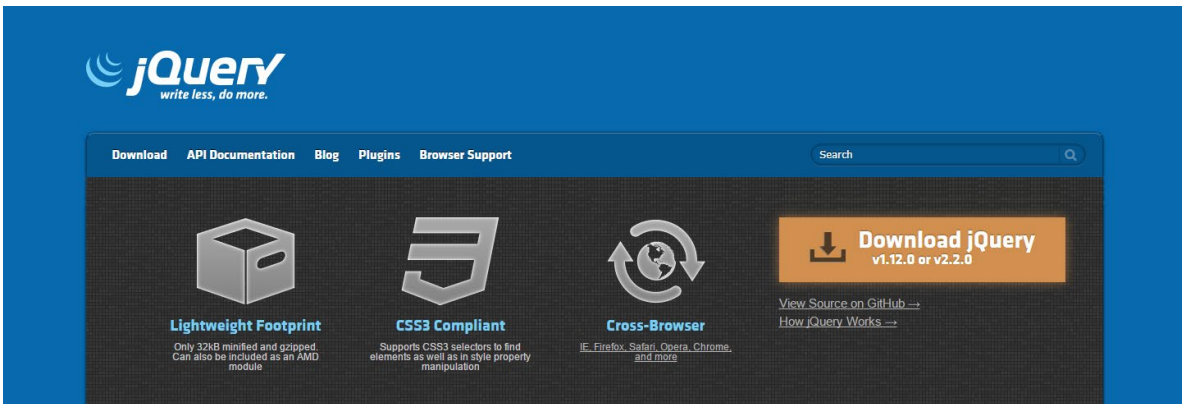
JavaScript developers are deluged with a simply astonishing amount of third party software. Even more interesting is the fact that **almost the entirety of these third party libraries is open source**. The net result is an environment in which it is extremely difficult for companies to pick a set of JavaScript tools on which to build the digital portion of their business.

There is also very strong sentiment behind all of the numerous frameworks. The rhetoric around which framework or library is the "right way" to build applications makes it even more difficult to find the signal in the noise. Furthermore, creators of frameworks and those that adopt those frameworks are hesitant to honestly critique their choice.

> I think there's a huge collective anxiety—a sort of sunk-cost fallacy—at play with web developers. We invest deeply in some tool, and so we're eager to justify, to ourselves and others, why the decision to use one tool over another was rational. I think this makes it hard to have a good dialog about how tools compare.
>
> **Brian Ford—Angular Team**

For this article, we asked several key developers in the community to help us gauge the future of some of the most popular third party libraries available. Some of these developers are the very authors of these frameworks, and some are implementers with a lot of experience on real world projects.

jQuery



Whether or not you find yourself a fan of jQuery, it cannot be denied that jQuery is still the most popular and widely used JavaScript library in the world. At the time these words are being typed, it is still the most starred repository on GitHub.

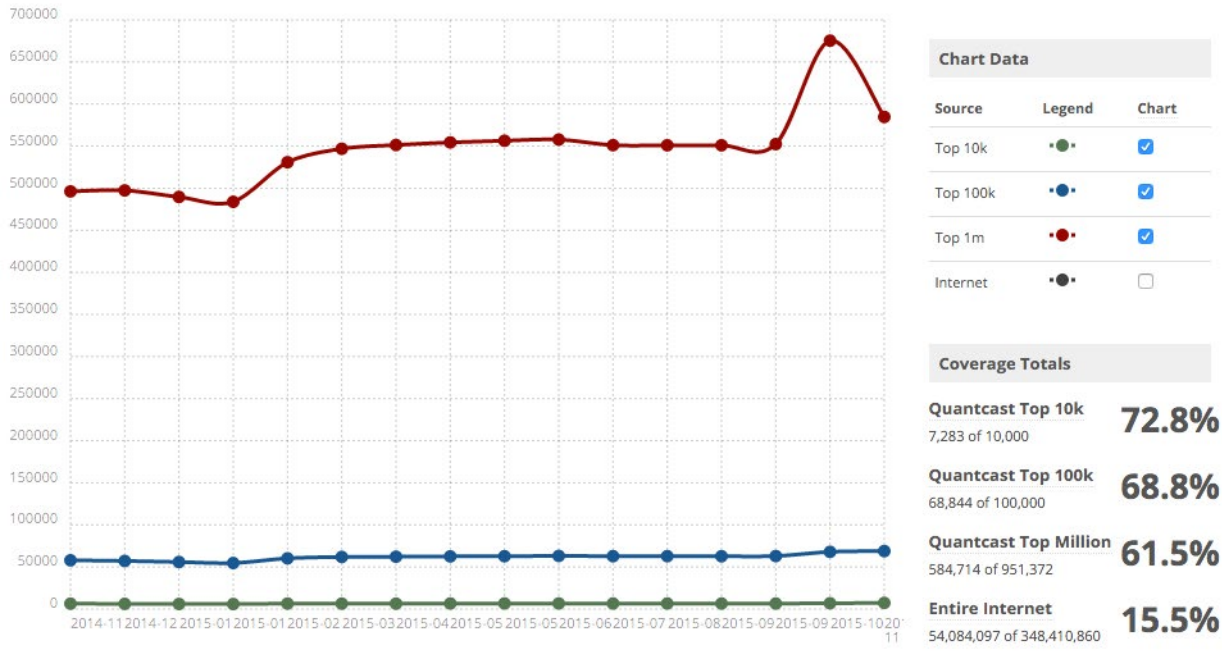
jQuery is now a decade old. In the last year, it has weathered some rather heavy criticism claiming that it is no longer necessary.

In his recent article, "The Relevance of jQuery: There And Back Again," Cody Lindley, author of "JavaScript Enlightenment, " writes that, "jQuery is as relevant today as it was when it was first written. " Lindley goes on to explain that this is not because developers need jQuery to perform basic DOM manipulation or XHR operations, but because jQuery provides a much cleaner API than native DOM code.

> "jQuery is simply a helpful library that you can optionally use when scripting HTML elements. And the fact is, most developers choose to use it when scripting the DOM because the API helps them get more things done with less code."
>
> **Cody Lindley: jQuery's Relevancy—There and Back Again**

jQuery predictions for 2016

The BuiltWith site shows a steadily growing usage of jQuery. There is no inclination that the use of jQuery is in decline. There is one jump in the top 1 million sites during the month of October, but the overall indication is a steady rise. Over 70% of the top 10K sites still use jQuery.



This is a clear indicator that jQuery usage will continue to rise during 2016. Evidence to suggest otherwise is anecdotal at best.

The level of hype around jQuery will continue to die, but the actual utility of the library will remain intact. Even newer frameworks and libraries offer some level of DOM abstraction. For instance, Angular 1.x uses jqLite, which is essentially the DOM selection and manipulation subset of jQuery. The API is rather identical to the actual jQuery library.

React

We would be remiss if we started our discussion of popular JavaScript libraries and we didn't open with ReactJS. The hype that React has garnered in just the past 12 months is nothing short of astounding. Add to this the list of big companies such as Netflix that are using React, and it cannot be denied that React has captured the hearts and minds of developers.

ReactJS is a far simpler library than its larger brothers and sisters such as Angular or Ember. However, it also does far less.

React is meant to the just the V in any MV* implementation. That means that React is primarily concerned with how visual components are built and rendered, and doesn't deal with the flow of data or the actual physical structure of the application. In order to achieve that, a full framework such as Flux or Redux must be added to the React equation.

The main features of React are the virtual DOM, and the way that it handles data binding. However, the appeal to developers is the large, high load applications that are already running on React. These would be applications like Facebook and Instagram. The appeal then becomes, “If it's good enough for Facebook, it should be good enough for my project too.”

React predictions for 2016

Expect the adoption of React to remain strong among large consumer applications. As developer Elijah Manor pointed out, React was picked for the Dave Ramsey Every Dollar project based on its existing use in large, well-known applications.

> "One compelling reason, and the reason we started using [React], is that we were launching a new product called EveryDollar. We were expecting a lot of load initially [...]. We wanted something that would scale well. React was really new, but it came out by Facebook and Instagram, and obviously, those are two applications that scale really well."
>
> **Elijah Manor—Polymorphic Podcast**

Developers can expect to see continued controversy around React's model of mixing markup with JavaScript, which will probably reach a peak point during 2016. While some developers like that the logic and markup for a component is contained in the same place, many have pointed out that it violates separation of concerns, which has long been a fundamental tenant of programming in general.

> "I see a willingness of many frameworks or libraries to assist developers in abandoning the authoring of SOLID code. Decades of learning should be taken seriously into account when you create a framework. React is again an example of this. We've learned hard lessons about not mixing JavaScript into our HTML. So, why now the eagerness to embrace mixing HTML into our JavaScript?"
>
> **Rob Eisenberg—Creator of Aurelia**

2016 will also be the year of commercial React components. While the React ecosystem is quite large, it is fragmented and is quickly going the way of jQuery plugins. Developers will begin to look for comprehensive solutions backed by a partner.

Enterprises will continue to watch React from a distance in 2016. The fact that React is such a small part of the overall application solution leaves developers to stitch together data access, routing and all of the other components needed for a full application stack. This will keep larger, more conservative shops from adopting React, in light of the lack of a complete story out of the box.

Developers should make it a point to learn React in 2016. It's likely that the model React has introduced will begin to bleed into other libraries and frameworks.

The React team did not return a request for input on this article.

AngularJS

The rapid rise of Angular during 2014 was remarkable to watch. It was remarkable because it was one of the rare times that the enterprise sector has so completely adopted an open source library. Angular seemed to answer so many outstanding questions for developers coming from more structured languages, such as Java or C#.

Angular has received some amount of criticism for its documentation and performance. Some feel that Angular is over-engineered and too complicated when compared with other frameworks. However, that has not stopped its adoption and legions of adoring fans.

The Angular ecosystem also grew drastically. Currently, there are 132,639 questions on StackOverflow that are tagged with Angular. Compare that with 6,969 for ReactJS and 19,031 for Backbone. There are several commercial UI libraries for Angular, including Kendo UI from Telerik.

Angular predictions for 2016

With the announcement of the impending Angular 2 release, developers can safely expect to see the full release of Angular 2 in the first quarter of 2016. Brian Ford from the Angular team explained the intentions behind Angular 2, and what was accomplished in 2015.

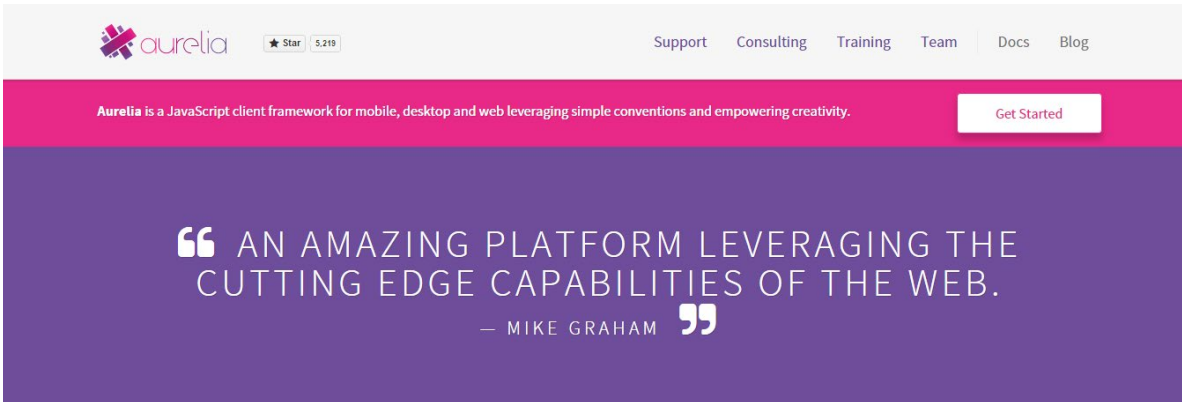
> "Our main motivation for Angular 2 is to do the things in Angular 1 that were impossible without significant breaking changes in Angular 1 apps. Mostly, we wanted to improve speed and robustness. We spent most of 2015 getting the core concepts in Angular 2 right. We took everything we learned from Angular 1, and from other open source projects in the same space (React, Ember, etc.). We're using a benchmark-driven approach. We're already as much as 10x faster than AngularJS 1, and we're only improving."
>
> **Brian Ford—Angular Team**

The Angular documentation and tooling will also improve significantly in 2015. The Angular team will spend a non-trivial amount of time helping developers to migrate from Angular 1 to Angular 2. “If you try to get started with Angular 2, you'll probably notice it has a lot of rough edges. We're working hard to make the experience as productive and enjoyable as possible,” says Ford. “A huge part of that [improving tooling, documentation and ecosystem] is going to be making migration a good experience, which will involve writing guides, building tools and making improvements to AngularJS 1.x to support migration.”

It's unlikely that swaths of developers will abandon Angular 1 for 2 in 2016, even if a smooth migration path is forged. Angular 1 will have a long tail, especially in the enterprise, where we will likely continue to see Angular 1 thrive in 2016.

Special thanks to Brian Ford from the Angular team for his input on this article.

Aurelia



Rob Eisenberg made Internet headlines in April of 2014 when he announced plans to join the Angular core team, with the intention of merging Durandal into Angular 2. He made headlines again in November when he announced that he would be leaving the Angular team to work on a JavaScript framework of his own, called Aurelia.

Aurelia tries to embrace existing/emerging standards to provide a full application framework. It relies heavily on ES6, a standards-based module loader and a component model that is compatible with the web components standard.

> “Rather than invent non-standard tech, or even buck against the web itself like some libraries do, we've chosen to embrace the web platform itself and to help developers use it to build future-compatible apps.”
>
> **Rob Eisenberg—Creator of Aurelia**

The most interesting thing about Aurelia is that it's possible to build applications using mainly ES6 classes, and not referencing the Aurelia framework at all. This goes a long way to make sure that applications are "future proof." When the next emerging standard arrives, or a different framework is used in the future, code written in Aurelia can be directly ported since it is primarily just ES6 classes.

Aurelia predictions for 2016

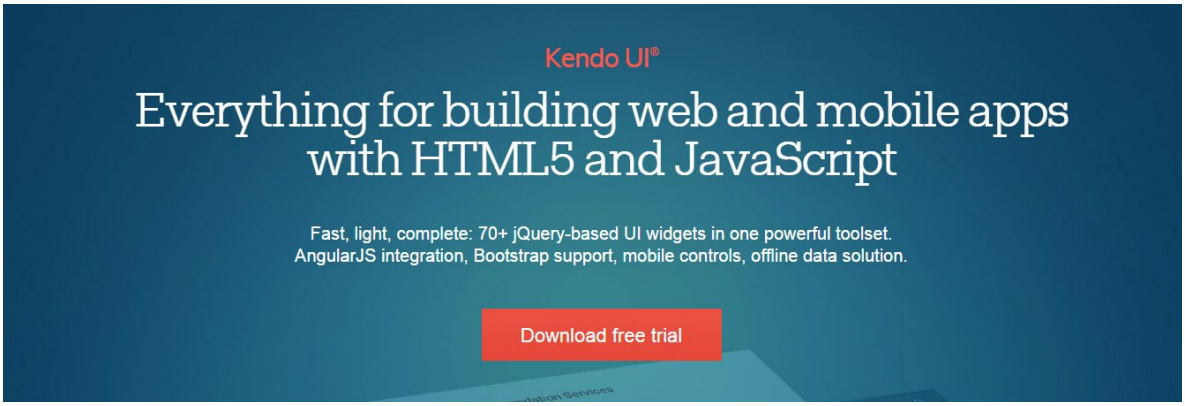
Developers will begin to adopt ES6 in droves starting in 2016. This will cause Aurelia to gain significant momentum, specifically in the .NET community and developers who are familiar with Eisenberg's previous work, such as Caliburn.Micro.

Aurelia has aggressive plans to be far more than a single-page app framework. "We've always seen Aurelia as a platform and ecosystem for building rich interactive applications on every device. In 2016, you'll see the next phase of that vision realized as we move beyond Aurelia's v1 release and on to other things we're planning," said Eisenberg. This may mean that Aurelia has plans to provide a similar JavaScript Native approach for mobile apps, similar to NativeScript and React Native.

More large enterprises will adopt Aurelia based on the fact that it is a supported product. So far, larger entities whose core business proposition is not technology have been slow to adopt JavaScript frameworks since they are for the most part supported by the community. Aurelia's model of having a core team of developers and offering support will cause many enterprises to choose it in favor of alternatives, specifically for the business partnership that Aurelia offers.

Special thanks to Rob Eisenberg from the Aurelia team for his input on this article.

Telerik Kendo UI



Kendo UI was launched in November of 2011, strictly as a set of jQuery based user interface components. At the time, developers were struggling with drastically different browser versions and feature support, as well as a wilderness of jQuery plugins that made it hard to assemble a cohesive application. Kendo UI was built on the premise that a developer could leverage a single set of UI components that would be guaranteed to work across all browsers, all the way back to IE 7.

While Kendo UI started as a UI library, it grew to include binding, routing, views, models and everything else developers needed for a full application solution. Despite offering those capabilities, developers continued to gravitate towards community standard frameworks, such as Angular, Ember and Durandal. However, they still wanted access to the robust widgets that Kendo UI offered.

Today Kendo UI is the largest open source jQuery-based UI library available. It is also a very successful commercial library, specifically in the areas of Data Grids, Schedulers, Data Visualization and Document Processing.

Kendo UI predictions for 2016

Kendo UI will begin to decouple itself from its own binding framework in 2016 to provide better integration with Angular, React, Ember, Aurelia and any other framework that developers prefer to use. Kendo UI will focus on being a set of UI components, and offer abstractions for the various larger frameworks so that it can be plugged in anywhere.

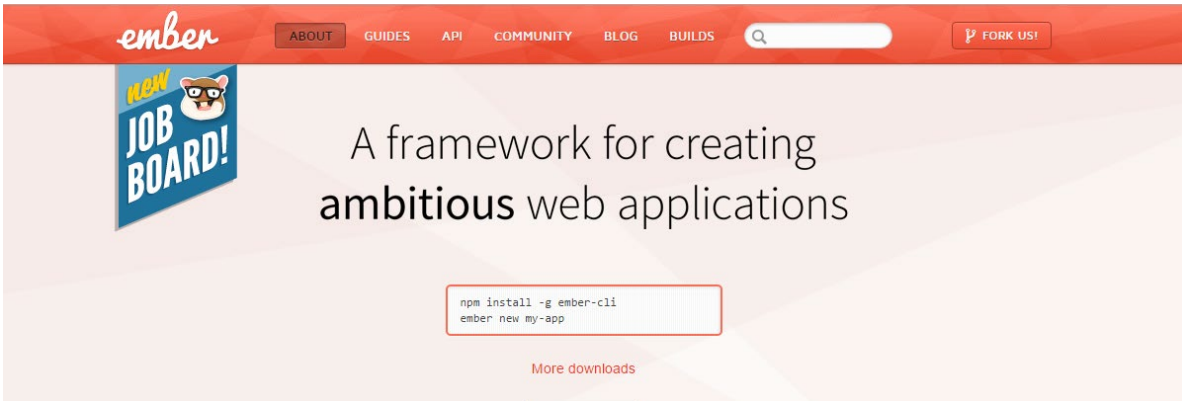
> "There are too many frameworks and not enough libraries. We need reusable, functional blocks that play well together."
>
> **Tsvetomir Tsonev—Kendo UI Team Lead**

Kendo UI will continue to invest heavily into extremely complex UI widgets, such as the Grid, PivotGrid and Spreadsheets widgets. This will entail adding new features and continuing to refine these components. Also, look for Kendo UI to ship additional complex widgets, such as an Interactive Timeline.

Kendo UI will also deliver more sample applications for other notable frameworks, such as Angular, Angular 2 and React.

Special thanks to Tsvetomir Tsonev from the Kendo UI team for his input on this article.

Ember



Ember has long been a staple of the JavaScript framework ecosystem. Grown from the remnants of SproutCore, it is an industry standard framework with many notable implementations. Although popular sites such as Discourse, Groupon, Vine and even the Apple Music desktop application use Ember, it does not get the same amount of attention as frameworks such as React. This may be due in part to its sheer age.

Ember also places a premium on future web standards. It has been an early adopter of many of the standards in future versions of JavaScript, such as promises. In addition, one of Ember's creators, Yahuda Katz, is on the TC39 committee, which is responsible for future versions of JavaScript.

Ember also tries to help developers write better code. This is done via use of a rigid and opinionated framework that seeks to guide developers into best practices, so that they fall into the “pit of success.”

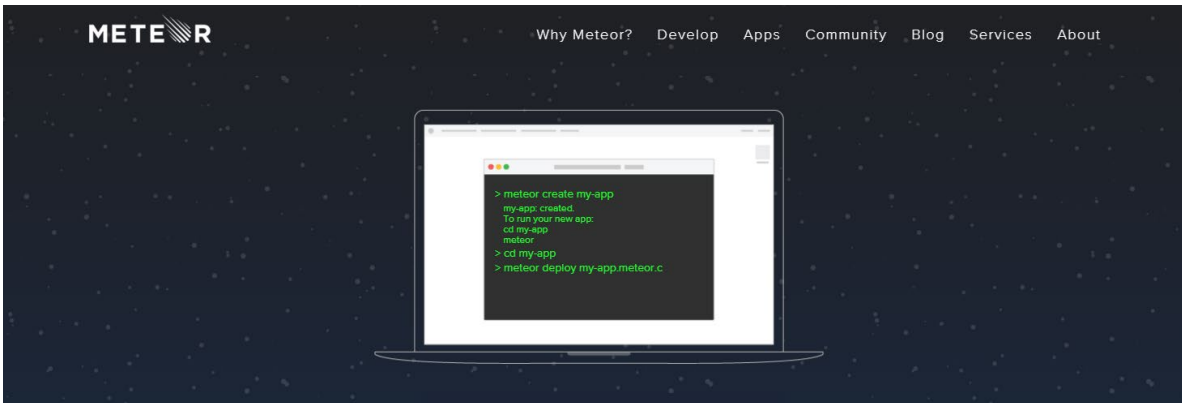
Ember predictions for 2016

Ember will continue to be a “sleeper” framework. While it won't be getting nearly as much airtime as its React and Angular peers, it will be the choice for large applications that need to service users at a massive scale. Look for other large sites to choose Ember for their next releases.

Ember will serve as the diametric comparison for React in terms of separating logic from markup. More and more developers will begin to draw this comparison in their own minds before deciding if they want to blend all of their component logic together as React prescribes, or separate it out as much as possible the way Ember dictates.

The Ember team did not respond to a request for input on this article.

Meteor



Meteor is another library that can't really be compared to your standard application frameworks such as Angular, Ember or Aurelia. While the aforementioned all do Universal or Portable JavaScript (server-side rendering), Meteor take the concept the rest of the way by providing both the server tier and database. It is a true application platform.

Meteor falls into the "Full Stack JavaScript" camp. It uses Node on the server-side, along with MongoDB. It uses an in-browser version of Mongo called MiniMongo to allow client-side code direct access to the data store. Developers are then free to choose if they want their code evaluated on the server, or the client.

The ecosystem around Meteor is currently centered around a Meteor proprietary package management system, Atmosphere. Developers can add the necessary meta data to their libraries so that they are compatible with the Atmosphere format. These libraries can then be pulled into any project with a simple command.

Meteor predictions for 2016

Meteor will begin to announce large-scale applications that currently use the platform. This will be the first time that many developers are aware of Meteor as they learn of some very high profile sites that are built on the technology.

Meteor will offer a free tier on its hosting platform, Galaxy. Right now, the cheapest package for Galaxy subscribers is a non-trivial $495 per month.

Look for Meteor to make a big announcement in 2016 regarding the mobile web. Currently, mobile support in JavaScript frameworks is not quite what it is on the desktop. The mobile web is still a far cry performance wise from native apps, and this is a gap that Meteor would like to close.

> "I believe that the biggest [gaps in existing frameworks] is true mobile support. We are still not there. There are many paths for [closing that gap] but it is still not a better experience than native apps."
>
> **Uri Goldshtein—Meteor Team**

Special thanks to Uri Goldshtein from the Meteor team for his input on this article.

Web Components

This wouldn't be a proper article without considering the implications on all of these frameworks when it comes to Web Components. Web Components are the technologies that are generally thought of as the emerging standards for creating interface components. However, Web Components can be used to create any piece of functionality that is possible with JavaScript, HTML and CSS.

In his article, "Why Web Components Aren't Ready For Production...Yet," TJ VanToll describes several of the major drawbacks to Web Components that have so far kept them from being adopted by developers. The main hang-up so far has been browser support.

> "The obvious reason to avoid Web Components is browser support. Although Web Components landed in Chrome 36, they only have partial support in Firefox, and they are not present in Safari or IE. Because cross-browser support won't be possible for a very long time, if it happens at all, a polyfill is a long-term necessity for developers that want to use Web Components outside of Chrome."
>
> **TJ VanToll—Why Web Components Aren't Ready For Production...Yet**

At one point, despite agreement from both Microsoft and Mozilla, Apple removed Shadow DOM from Safari altogether. Unfortunately, since users have no choice of browsers on iOS, Mobile Safari currently holds the rest of the web hostage. If a technology is not supported in Mobile Safari, it's likely that it will not be used by developers, especially if it's as hard to polyfill as Web Components are.

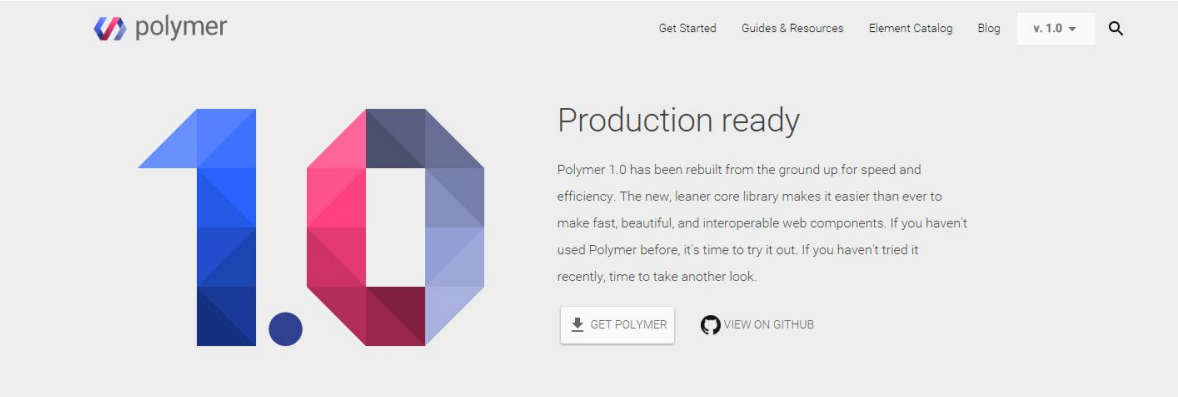
Web Components predictions for 2016

All major browsers will support Web Components by the second half of 2016. This prediction is based on the recent addition of Shadow DOM to WebKit.

All JavaScript frameworks will begin to swap out their own technology for Web Components standards as those standards become widely supported.

> "Web Components" is an umbrella term for a bunch of upcoming APIs, and there's more than one way to break apart your app into smaller parts. The general idea is for Angular to pragmatically adopt new browser features as they become more available."
>
> **Brian Ford—Angular Team**

Polymer



The Polymer project from Google is often mistaken for, and used interchangeably with Web Components. Polymer is an application framework that is built on the concept of Web Components, and attempts to polyfill certain Web Components APIs (such as Shadow DOM) for browsers where those APIs don't exist.

Despite Google throwing what appears to be the entirety of their influence into the Web Components arena, Polymer has not yet garnered mainstream support. This is most likely due to the same browser compatibility issues that Web Components suffer from.

Google has created an impressive list of components for Polymer that encompasses UI, animation and even seamless integration with Google's own APIs. Developers looking to adopt Polymer will find virtually everything they need for their applications, especially if they are interested in doing Material Design.

At the 2015 Chrome Dev Summit, Google announced that there are now 1 million sites running on Polymer. These include some big name companies, such as GE. It also includes many of Google's own internal properties, such as Google Play Music.

Polymer predictions for 2016

Expect to see Google transition the majority of its applications over to Polymer in 2016. Google obviously sees Web Components as a strategic advantage. Given that it has little developer adoption, it is clear that Google will continue to push Web Components and Polymer by adopting it internally.

Google will release application framework, routing and internationalization components for Polymer in 2016. These are already on the roadmap.

Web Components will begin to take hold in the development community during the second half of 2016. This will cause an identity crisis for web developers who have been coding to frameworks for so long.

Expect to see the first batch of commercial Web Components by the later months in 2016.

The Polymer team did not respond to a request for input on this article.

Final developer predictions

Lastly, here are some generic predictions for developers that I gathered based on my own research and conversations with those who provided input for this article.

- Universal/Portable JavaScript will be big in 2016, seeing as how React, Angular 2, Meteor, Ember and Aurelia all support it.
- Developers will be expected to know ES6. Learn it.
- Browser compatibility will largely be a non-issue, based on the fact that fundamental polyfilling has essentially been mastered.
- JavaScript frameworks will begin to target more than the web. This is already happening with React Native and NativeScript. Expect Ember and Aurelia to get in this game as well.

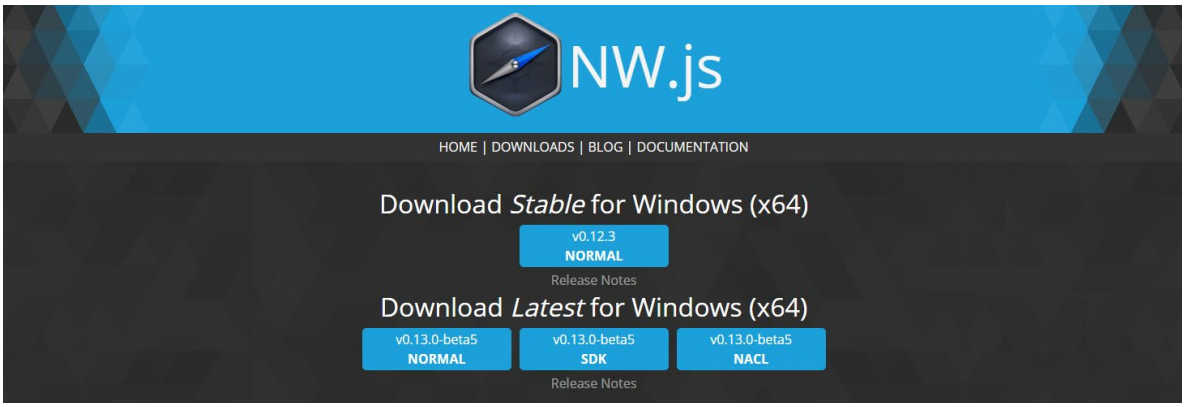
JAVASCRIPT'S NEW FRONTIERS

In the last several years JavaScript, a scripting language designed for use in web browsers, has been used in an increasingly diverse set of software applications. With JavaScript now running as server-side code, driving iOS and Android apps, and even controlling robotics, it's hard to find a software ecosystem that JavaScript hasn't influenced.

Part of what's driving JavaScript's expansion into these "new frontiers" is performance. Whereas years ago running JavaScript on a server was unthinkable, Google's entry into the browser and JavaScript engine world in 2008 sparked a performance competition that drastically improved the speed of the language. More recent efforts such as asm.js have only furthered this effort.

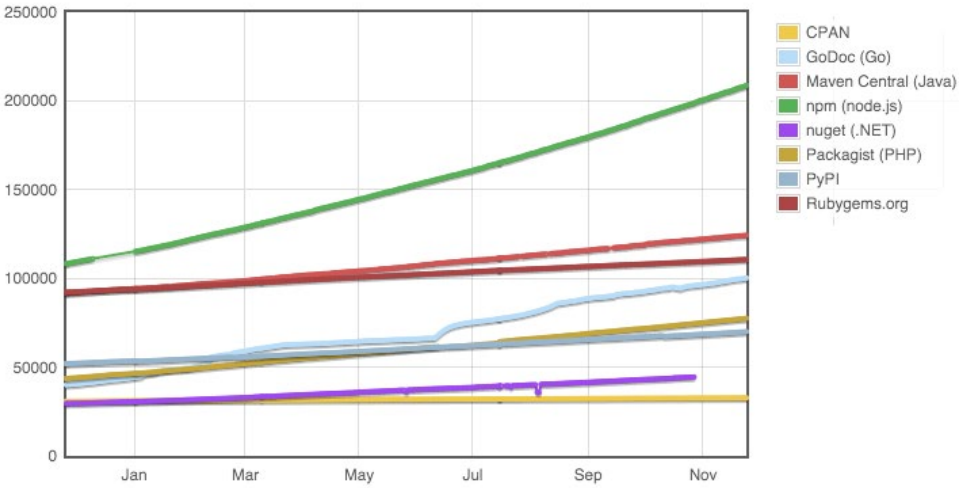
In this article, we'll look at what's next for the JavaScript frameworks being used to run server-side JavaScript, build mobile apps and desktop applications. We'll get the perspective from many key developers involved in building these solutions directly. Let's start by looking at what is perhaps JavaScript's first new frontier: Node.js.

Node.js



Node.js is an open source runtime environment based on Google's V8 JavaScript engine. Although plenty of companies and frameworks had tried to run JavaScript on the server, Node.js was the first runtime to succeed at doing so at scale.

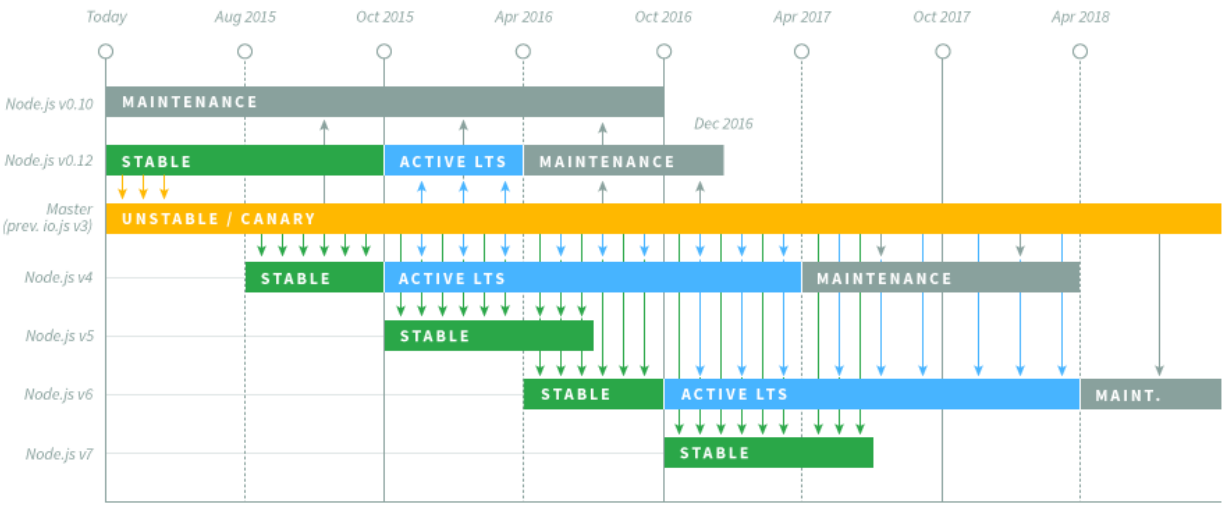
Node.js was first written in 2009, and has since skyrocketed in popularity. The list of companies using Node.js is enormous, and the recently formed Node.js Foundation includes the likes of IBM, Intel, PayPal and Microsoft. The Node.js package manager, npm, became the biggest package manager in the software world in 2014, and now contains nearly twice as many modules as similar package managers from the Java and Ruby worlds.



Growth of npm as a package manager. Image from modulecounts.com.

However, the success of Node.js hasn't come without growing pains. In late 2014, a group of developers forked the popular framework, citing the lack of active and new contributors and the lack of releases. The new framework, io.js, quickly gained followers and community support, leaving many to fear a long-term fragmentation in the Node.js world. Thankfully, those fears were averted when Node.js and io.js merged in June 2015.

Part of the merger involved the formation of an LTS, or a Long-Term Support plan for Node.js releases. Under the plan, Node.js will designate one release per year an LTS release, and will actively support that release for a full 18 months.



The development cycle is aimed to appease both developers that want to stay on the cutting edge, as well as large companies that need a stable release they can count on for years to come. And the development cycle has major implications for the future of Node.js. When I asked Mikeal Rogers from the Node Foundation what the biggest change coming for Node.js in 2016, he had this to say:

> "The new development cycle is going to be the biggest change. We'll have two major releases each year, with only one of those releases receiving Long Term Support. That's significantly more than before and we've never truly had LTS before so this is all a big change for developers and a new opportunity for enterprises to expand adoption as well."
>
> **Mikeal Rogers, Node Foundation**

Node.js in 2016

In 2016, expect to see further adoption of Node.js and its package manager npm. The continued adoption of Node from large companies—Microsoft, IBM, Intel, Progress, etc.—as well as enterprise-friendly features such as long-term support plans, may signal a growth in Node.js adoption in the enterprise—replacing typical enterprise solutions like .NET and Java.

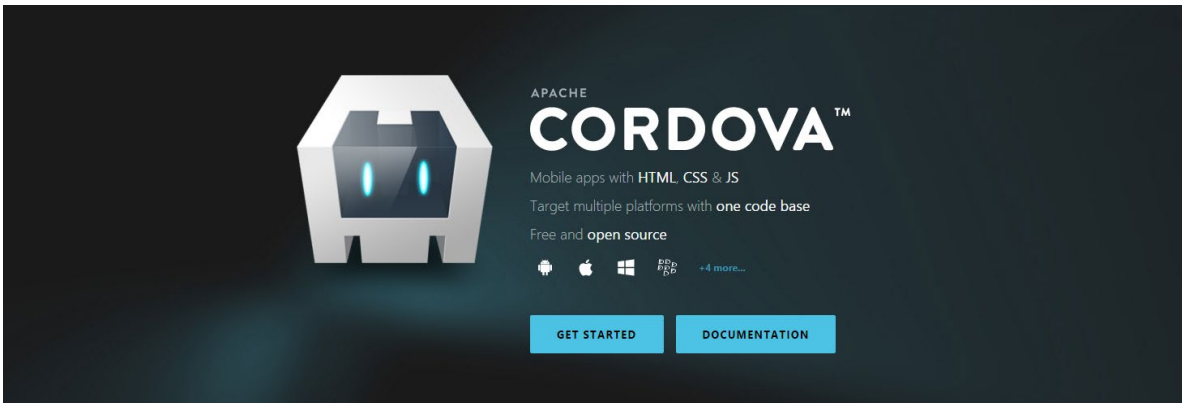
Expect the exponential growth in modules on npm to continue as well, as npm's recent releases have aimed at providing better support for client-side JavaScript, thus replacing the need for client-side JavaScript package managers such as Bower. As developers start registering their client-side scripts and jQuery plugins on npm, npm's reach will only grow. In fact, according to Mikeal Rogers, the major reason for npm's growth is that it is an ecosystem of ecosystems.

> "A few years back I quantified the growth rate of npm and created a predictive graph. At the time people thought it was insane, because it said that in a little over a year we'd have over 100K modules and that the rate of growth wouldn't level out. We hit 100K modules within a few days of what we predicted, which even I was shocked by.
>
> If you dive deep into npm's growth, you'll see that what is pushing it forward is that npm is an ecosystem of ecosystems. It's the best place to build sub-platforms for a variety of use cases. Parts of that ecosystem are leveling off but they keep getting replaced by new, rapidly growing ecosystems."
>
> **Mikeal Rogers, Node Foundation**

As evidence of this claim, basically every other JavaScript solution in this article—including Cordova, React Native and NativeScript—all use npm as a package manager. A quick npm search for "jquery", "polymer", "meteor", or "react" can give you an idea of the sheer scale npm operates at now. As JavaScript grows in popularity, npm grows in popularity. And as npm grows in popularity, so does Node.js. The future looks bright for the software world's first mainstream server-side JavaScript framework.

Let's shift our focus to some technologies that don't run JavaScript on the server, but rather, use JavaScript to drive mobile apps.

PhoneGap and Cordova



Much like Node.js was the first mainstream solution for running JavaScript on the server, PhoneGap was the first mainstream solution for using JavaScript to run native mobile apps. PhoneGap was originally created by Nitobi in 2009, and was acquired by Adobe in October 2011. As part of the acquisition, the PhoneGap source code was donated to the Apache Software Foundation and the project became known as Cordova. Today, Cordova is a free and open source framework that many companies contribute to, and PhoneGap is an Adobe-owned distributor of Cordova.

Over the years, Cordova has defended itself against a perception of bad performance, with the most notorious complaint coming from one of technology's most influential people in 2012.

> "When I'm introspective about the last few years, I think the biggest mistake that we made as a company is betting too much on HTML5 as opposed to native... because it just wasn't there."
>
> **Mark Zuckerberg, Facebook**

Since 2012, a number of companies have stepped in to attack this performance problem. This includes performance-minded UI frameworks like Ionic, Onsen, and Kendo UI Mobile, tooling improvements from Telerik and the PhoneGap team, new web views such as those provided by Crosswalk, and high-quality plugins, such as those found in the Telerik Verified Plugins Marketplace.

The Telerik Verified Plugins Marketplace

In addition to company provided performance aids, new features provided by mobile OS makers, such as Android's auto-updating web views and iOS's new WKWebView API, have greatly improved the Cordova performance situation. With that in mind, I asked Brian LeRoux of Adobe about what's next for Cordova.

> "Cordova has grown very deliberately stable. We strive to keep things simple, push the features out to the plugins interface, and stay on top of platform upgrades like Android M and iOS 9 as much as possible. It took a few years of thrashing, but 'small modules' mindset is beginning to take hold which makes me happy. The end dev audience won't see this unless they extend Cordova with their own distribution."
>
> **Brian LeRoux, Adobe**

Cordova in 2016

Much like Node.js, Cordova's stability will appeal to large companies, many of which are just dipping their toes into the waters of mobile development. The Cordova approach to building mobile apps with HTML, CSS and JavaScript will continue to appeal to web developers, especially when compared to native development options involving Xcode and Android Studio.

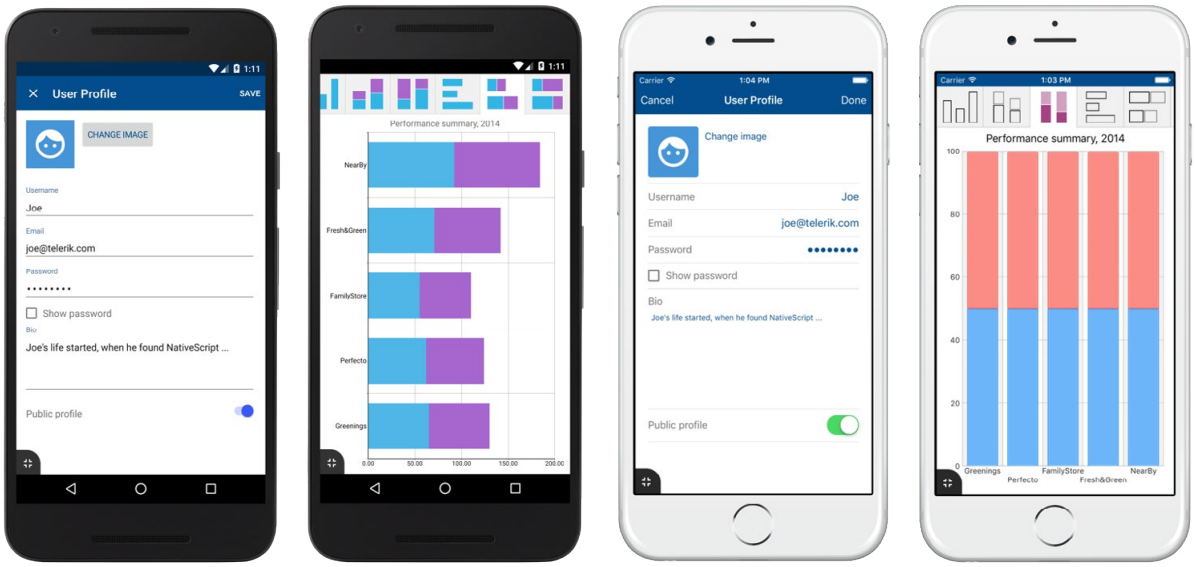
Although Cordova continues to grow in popularity, its development approach is being challenged from two different angles. The first is from Google, who is pushing the concept of progressive apps, or true web apps with native-like features such as splash screens, home screen placement and offline access. Progressive apps are still in their early days, and their features are still only offered on Chrome for Android, however, expect Google to continue pushing the concept of progressive apps in 2016.

The bigger immediate challenge to Cordova development comes from a recent development in the JavaScript world: using JavaScript to build truly native mobile applications.

Native mobile apps

The year 2015 saw the emergence of a new category of JavaScript-based mobile app development known as "JavaScript Native." Unlike Cordova- and PhoneGap-based apps, JavaScript Native apps use a platform's native controls and paradigms to build their user interfaces; there is no browser or web view involved.

JavaScript Native frameworks attempt to offer a best-of-both-worlds way to build iOS and Android apps: use JavaScript to write your application logic (rather than Java, Swift and so forth), and use a platform's native user interface APIs to build apps that fit in on the native OS and offer the best possible performance.



Example of mobile apps built with JavaScript. Check out the source code.

React Native and NativeScript were the first JavaScript Native frameworks to publicly release in 2015; followed by others, such as Fuse and tabris.js. Different frameworks offer different features—for instance React Native lets you reuse the React JavaScript framework, NativeScript lets you access iOS and Android APIs directly from JavaScript, and so forth—but they share the high-level approach to building truly native apps with JavaScript.

Although the idea of building native apps with JavaScript can sound appealing to web developers, JavaScript Native frameworks can have some drawbacks when compared to frameworks like Cordova. Here are a few:

- Because JavaScript Native frameworks don't use a browser, you have to learn framework-specific APIs for building your interfaces, rather than simply using HTML as you would in a Cordova app.
- Because JavaScript Native apps are native apps, memory management can be a concern in larger apps, just like it is in native iOS and Android apps.
- Finally, JavaScript Native frameworks are new, and as such, there are fewer examples and tutorials. The frameworks themselves are less mature than frameworks that have been under active development for many years.

I asked Christopher Chedeau (aka Vjeux) from the React Native team, and Valio Stoychev, NativeScript's product manager, about what's coming for their frameworks in 2016, and both echoed this focus on stability.

> "For React Native, we exited the phase where it was a crazy idea/prototype and now enter the phase where we need to make it solid. You should see a lot of work being done on performance tooling/optimizations, improvement of all the core APIs, better error messages, fix edge cases... This way engineers at Facebook and outside can build the high quality mobile apps they want to."
>
> **Christopher Chedeau (Vjeux), Facebook**

> "As our user base rapidly grows, we need to make sure our users have a robust framework they can count on for building real-world applications. Therefore we intend to continue working on things like performance and debug tooling to improve the NativeScript developer experience. Our other major focus is our work with the Angular 2 team, which we anticipate will continue throughout 2016."
>
> **Valio Stoychev, Telerik**

JavaScript Native in 2016

Expect 2016 to be the year of stability and adoption for JavaScript Native apps in 2016. As frameworks like React Native and NativeScript solidify their feature set, expect to see an increase in tooling being created around those frameworks, such as Telerik's own Telerik Platform for building NativeScript apps.

Time will tell whether the hype JavaScript Native apps generated in 2015 will transfer to large-scale usage in 2016, but the number of high-quality applications already being built with these frameworks (see React Native's showcases and NativeScript's showcases) suggests that the JavaScript native approach to building applications will be around for some time to come. For companies that need native apps with native UIs,

JavaScript Native frameworks offer a compelling option when compared to building an iOS app with Xcode and Objective-C/Swift, as well as Android Studio and Java—especially considering many companies have developers with existing JavaScript development skills.

Overall, JavaScript Native apps represent an exciting new frontier for JavaScript developers. No longer do JavaScript developers have to learn native programming languages to write a native mobile app. And native mobile apps aren't the only software world JavaScript is creeping into—the same is true of traditional desktop applications.

Desktop apps

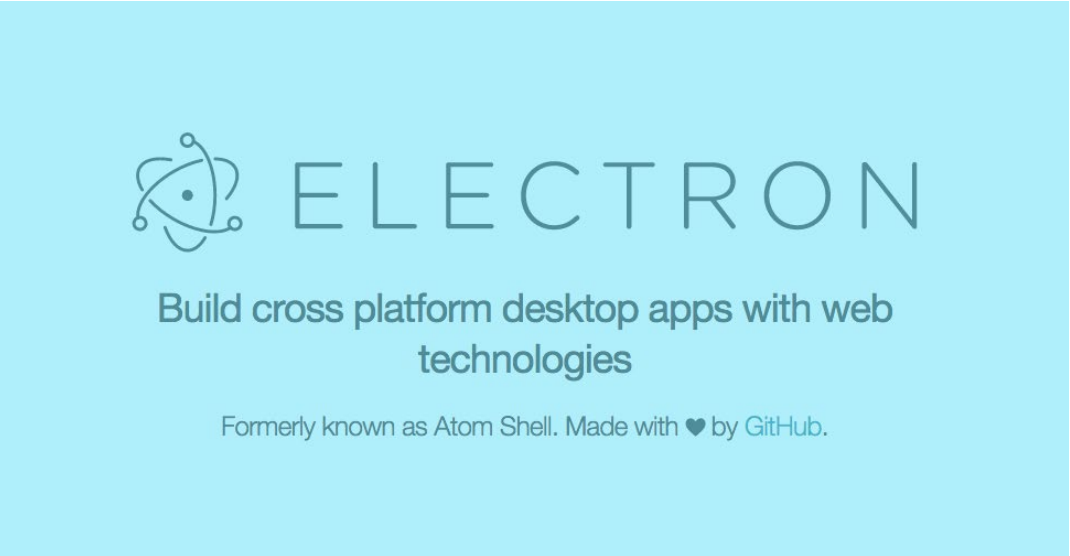
Traditionally, if you wanted to build a Windows or Mac app you'd use platform-specific tools like WPF & Windows Forms, or cross-platform interfaces using something like Java or Adobe Air. But, like every other software ecosystem discussed in this article, JavaScript-based solutions are slowly working their way into this picture.

The first JavaScript-based solution in this space was Node-WebKit, which was created by Intel and was open sourced in late 2011. Node-WebKit, now called NW.js because of its switch from WebKit to Chromium internally, works somewhat similarly to Cordova, but for desktop apps.

NW.js was developed at Intel, and released back in 2011

NW.js packages up a web application in a native shell, while providing access to native desktop APIs, such as the file picker, window menus and so forth. The combination lets you write Windows, OS X and Linux desktop applications using standards-based web technologies.

Fast forward a few years and NW.js is not the only framework using such an architecture. In April 2015, GitHub announced Electron, a similar framework for building cross-platform desktop apps.



GitHub's Electron was released in April 2015

Electron was developed as the shell for Atom, GitHub's web-based text editor, and has since been decoupled to use in any project. With GitHub's backing, Electron has surged in popularity, and now has over 20,000 starts on GitHub (quickly catching up with NW.js's 25,000+ starts). Electron also made headlines in 2015 as the engine behind Microsoft's new cross-platform Visual Studio Code IDE, and a quick look through a list of community-created Electron resources shows just how popular Electron has become in the development community.

Desktop apps in 2016

Like many of the technologies discussed in this article, the future seems bright for these cross-platform JavaScript-based tools for building desktop apps. With the likes of GitHub, Microsoft and even Slack—which isn't built on NW.js or Electron, but also takes the approach of using web technologies to build a native app—other companies can feel confident building desktop apps with web technologies. Expect projects like NW.js, Electron and others to drive many of the new desktop applications you'll see launching in 2016.

JavaScript's new frontiers in 2016

Although this article has discussed a seemingly disparate set of topics—server-side code, mobile apps and desktop apps—the narrative has been the same: in the span of a few years, running JavaScript in these contexts has gone from unthinkable to mainstream. In less than a decade, JavaScript has gone from a toy language for handling image rollovers, to perhaps the world's most popular programming language—and there seems to be no end to where JavaScript can go.

In 2007, Jeff Atwood famously stated that, "any application that can be written in JavaScript, will eventually be written in JavaScript," and that statement has never seemed more prophetic. JavaScript has reached places where there wasn't space enough to cover in this article, such as running on hardware through projects like Johnny-Five, and being offered as a first-class citizen for building native apps on Apple's recently announced tvOS for Apple TV.

One of the reasons driving JavaScript's growth is the desire for a single development model to build software for multiple paradigms. Companies, especially small companies, cannot possibly hire developers with the expertise to reach the crazy number of operating systems and devices people use today. This is even a problem at a company at Facebook's scale, as Christopher Chedeau shares:

> "To me, the big tragedy of the developer world today is that communities are divided by language, we even call them ecosystems. JavaScript, Java, Objective-C, Python, C++, <name your favorite language>. What happens is that there is a massive waste of effort as each ecosystem has the same tools such as package manager, IDE, core libraries, knowledge base..."

To give a concrete example, at Facebook, we need to implement the exact same feature three times: for Web, iOS and Android. Even worse, because it's so hard for one engineer to get ramped up in those ecosystems, we usually have three people implementing that feature. This is sad.

> "In order to solve that, my intuition is that there needs to be a single language/ecosystem. With React Native, we opted for JavaScript, but in the grand scheme of things it doesn't matter which language it is. What's most important is that there is only one."
>
> **Christopher Chedeau, Facebook**

With JavaScript rapidly becoming a viable option in all of these worlds—mobile, desktop, server, hardware—it's uniquely positioned to make this desire to build once a reality. Time will tell whether JavaScript's meteoric growth continues in 2016 and beyond, but the surge in popularity of JavaScript tooling across software ecosystems seems to indicate that there's no end in sight.

With that in mind, I'll let Brendan Eich's famous quote stand as the last word in this article: Always bet on JS.

ABOUT THE AUTHORS

Burke Holland

Burke Holland is a web developer living in Nashville, TN and the Director of Developer Relations at Telerik. He enjoys working with and meeting developers who are building mobile apps with jQuery/HTML5 and loves to hack on social API's. Burke works for Telerik as a Developer Advocate focusing on Kendo UI. You can follow him on Twitter at @burkeholland.

Cody Lindley

Cody Lindley is a front-end developer working as a developer advocate for Telerik focused on the Kendo UI tools. He lives in Boise, ID with his wife and three children. You can read more about Cody on his site or follow him on Twitter at @codylindley.

TJ VanToll

TJ VanToll is a developer advocate for Telerik, a jQuery team member, and the author of jQuery UI in Action. TJ has over a decade of web development experience—specializing in performance and the mobile web—and speaks about his research at conferences around the world. TJ is @tjvantoll on Twitter and tjvantoll on GitHub.