



Future of JavaScript in 2017 and Beyond

WHITEPAPER

JavaScript's Journey Through 2016

Every year, there seems to be more and more ways to use JavaScript and 2016 turned out to be no different. Depending on your level of optimism, this can be extremely exciting or extremely [confusing](#). Last year, we made some predictions about JavaScript in 2016. Now, we'll look back to see if our predictions held true or went the way of the political pundits. Then, we'll use what we've learned this year to make an educated guess on what we'll get out of 2017.

**PREDICTING THE SUCCESS OR FAILURE OF A NEW PRODUCT
BASED ON WHAT ENGINEERS AND PROGRAMMERS ARE SAYING ABOUT IT.**

IF THEY SAY...	IT MEANS...
"IT DOESN'T DO ANYTHING NEW"	THE PRODUCT WILL BE A GIGANTIC SUCCESS.
"WHY WOULD ANYONE WANT THAT?"	
"REALLY EXCITING"	THE PRODUCT WILL BE A FLOP. YEARS LATER, ITS IDEAS WILL SHOW UP IN SOMETHING SUCCESSFUL.
"I'VE ALREADY PREORDERED ONE."	
"WAIT, ARE YOU TALKING ABOUT <UNFAMILIAR PERSON'S NAME>'S NEW PROJECT?"	THE PRODUCT COULD BE A SCAM AND MAY RESULT IN ARRESTS OR LAWSUITS.
"I WOULD NEVER PUT <COMPANY> IN CHARGE OF MANAGING MY <WHATEVER>."	WITHIN FIVE YEARS, THEY WILL.

This comic doesn't do anything new.

ES2015 Browser Implementation

As of June 2016, the development community moved from the 6th edition of ECMAScript—[ES2015](#) (once referred to as ES6)—to the 7th edition—[ES2016](#). The next edition developers will move to is [ES2017](#) but that may not arrive until June 2017 (if not later). Sometimes, like in the chart below, the upcoming features of ES2016 and ES2017 are just lumped into the ES2016+ category. This whitepaper will refer back to this chart, so for clarity sake ES2016+ will be used to describe future features as well. ES2015 contained a lot of features that developers felt would make programming in JavaScript better, like arrow functions, promises, destructuring and more. Here we'll look into how the features of ES2015 were handled in 2016 and if they held up to our expectations.

In order to use features from ES2015 that weren't supported by browsers, servers or runtimes source-to-source transpilers like [Babel](#) and [Traceur](#) are used so developers can write with the ES syntax and then have it compiled into compatible JavaScript. Thanks to the ever-handy [ECMAScript compatibility table](#) from [kangax](#), we can see a lot of green, in the chart—this means that many of the features are supported in various browsers.

Currently, Firefox 50 is supporting 92% of the ES2015 features, Chrome 55+ and Node 6.5+ are at 97%, while Safari 10 and iOS 10 are at 100%. The only feature holding back the 97%-ers is the optimization feature of ES2015, [proper tail calls](#). This is great news for developers who want to use these features without transpilers.

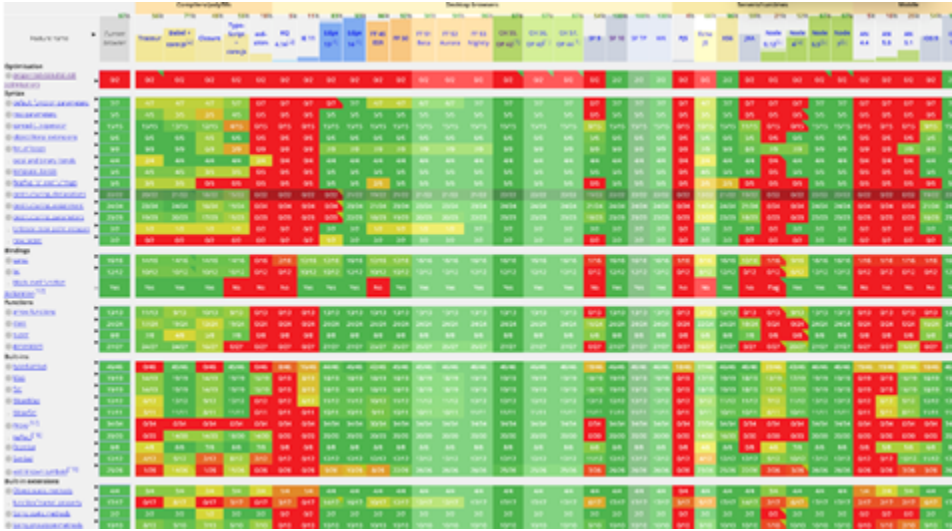
The chart displays the following categories and features:

- Opera:** 97.0, 96.0, 95.0, 94.0, 93.0, 92.0, 91.0, 90.0, 89.0, 88.0, 87.0, 86.0, 85.0, 84.0, 83.0, 82.0, 81.0, 80.0, 79.0, 78.0, 77.0, 76.0, 75.0, 74.0, 73.0, 72.0, 71.0, 70.0, 69.0, 68.0, 67.0, 66.0, 65.0, 64.0, 63.0, 62.0, 61.0, 60.0, 59.0, 58.0, 57.0, 56.0, 55.0, 54.0, 53.0, 52.0, 51.0, 50.0, 49.0, 48.0, 47.0, 46.0, 45.0, 44.0, 43.0, 42.0, 41.0, 40.0, 39.0, 38.0, 37.0, 36.0, 35.0, 34.0, 33.0, 32.0, 31.0, 30.0, 29.0, 28.0, 27.0, 26.0, 25.0, 24.0, 23.0, 22.0, 21.0, 20.0, 19.0, 18.0, 17.0, 16.0, 15.0, 14.0, 13.0, 12.0, 11.0, 10.0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0
- Firefox:** 50.0, 49.0, 48.0, 47.0, 46.0, 45.0, 44.0, 43.0, 42.0, 41.0, 40.0, 39.0, 38.0, 37.0, 36.0, 35.0, 34.0, 33.0, 32.0, 31.0, 30.0, 29.0, 28.0, 27.0, 26.0, 25.0, 24.0, 23.0, 22.0, 21.0, 20.0, 19.0, 18.0, 17.0, 16.0, 15.0, 14.0, 13.0, 12.0, 11.0, 10.0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0
- Safari:** 10.0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0
- Chrome:** 55.0, 54.0, 53.0, 52.0, 51.0, 50.0, 49.0, 48.0, 47.0, 46.0, 45.0, 44.0, 43.0, 42.0, 41.0, 40.0, 39.0, 38.0, 37.0, 36.0, 35.0, 34.0, 33.0, 32.0, 31.0, 30.0, 29.0, 28.0, 27.0, 26.0, 25.0, 24.0, 23.0, 22.0, 21.0, 20.0, 19.0, 18.0, 17.0, 16.0, 15.0, 14.0, 13.0, 12.0, 11.0, 10.0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0
- Edge:** 15.0, 14.0, 13.0, 12.0, 11.0, 10.0, 9.0, 8.0, 7.0, 6.0, 5.0, 4.0, 3.0, 2.0, 1.0
- Node:** 6.5.0, 6.4.0, 6.3.0, 6.2.0, 6.1.0, 6.0.0, 5.12.0, 5.11.0, 5.10.0, 5.9.0, 5.8.0, 5.7.0, 5.6.0, 5.5.0, 5.4.0, 5.3.0, 5.2.0, 5.1.0, 5.0.0, 4.8.0, 4.7.0, 4.6.0, 4.5.0, 4.4.0, 4.3.0, 4.2.0, 4.1.0, 4.0.0, 3.10.0, 3.9.0, 3.8.0, 3.7.0, 3.6.0, 3.5.0, 3.4.0, 3.3.0, 3.2.0, 3.1.0, 3.0.0, 2.10.0, 2.9.0, 2.8.0, 2.7.0, 2.6.0, 2.5.0, 2.4.0, 2.3.0, 2.2.0, 2.1.0, 2.0.0, 1.10.0, 1.9.0, 1.8.0, 1.7.0, 1.6.0, 1.5.0, 1.4.0, 1.3.0, 1.2.0, 1.1.0, 1.0.0

ES2015 Compatibility chart from [kangax](#)

Although there are some features of ES2016+ already being supported (Firefox 52+ is already at 91%) there is still a lot of red.

So far though, this feature list is shorter than ES2015 and the only “large feature” is async functions (which is already supported on Firefox 52+ and Chrome 55+). With the success of the browser support for ES2015, it looks very likely developers will once again get close to 100% compatibility with the main browsers for the ES2016+ features.



ES2016+ Compatibility chart from [kangax](#)

Modules, the Most Important Addition?

Last year, we asserted that ES6 modules would be the most important addition. As predicted, a lot of developers have taken advantage of the ES6 module syntax in their code thanks to transpilers like Babel or Traceur. It’s hard to gather numbers to back this up but if you read all the articles on the “top” or “favorite” new features from ES2015, the modules feature is almost always listed.

We were hoping for a native module system in 2016 so that commonJS, AMD, UMD, and non-native loaders like browserify, webpack and systemJS were no longer necessary. However, tackling the loading process for modules seems to be quite a feat, so that feature may still be far off. That said, there does seem to be a good measure of interest and work going into this feature. There is [a great recap](#) from James Snell about a [TC-39](#) meeting he attended to get info on the feature. “There is a proposal being put before TC-39 that

© 2017 Progress. All Rights Reserved.

would introduce a new `import()` function.” Snell wrote. The `import()` function...is processed at evaluation. It also imports an ESM (or CommonJS module) but, like the `require()` method in Node.js currently, operates completely during evaluation. Unlike `require()`, however, `import()` returns a Promise, allowing (but not requiring) the loading of the underlying module to be performed fully asynchronously.”

This feature would also enable developers to make calls like `await import('foo')`. To be clear, this may still be a long way off but it is currently [in stage 3](#). Here’s a breakdown of their process to explain what ‘stage 3’ actually means: [TC-39 Process](#). Based on the amount of interest in this feature, it’s likely that it will get released in 2017.

Popular Features of 2016

Following ES6 modules, we predicted that the other stand-out feature of the year would be promises. No one likes [callback hell](#) or the [pyramid of doom](#), so the addition of native promises was a very welcomed move. Promises have been covered in a lot in posts and tutorials across the web, likely because they seem complex

The screenshot shows the GitHub repository page for Ecma TC39. The repository is titled "Ecma TC39" and is described as "Ecma International, Technical Committee 39 - ECMAScript". It has a yellow "JS" logo. The page shows a list of repositories under the "Repositories" tab, with filters for "Type: All" and "Language: All". The repositories listed are:

- ecma262**: Status, process, and documents for ECMA262. Language: HTML. 3,488 stars, 237 forks. Updated 14 minutes ago.
- test262**: Official ECMAScript Conformance Test Suite. Language: JavaScript. 405 stars, 132 forks. Updated 5 hours ago.
- ecma402**: Status, process, and documents for ECMA 402. Language: HTML. 54 stars, 12 forks. Updated 7 hours ago.

On the right side, there is a "Top languages" section showing HTML, JavaScript, and CSS. Below that is a "People" section showing 23 profile pictures of contributors.

The brave ECMA TC-39 <https://github.com/tc39>

when developers first start using them.

A few other ES6 features that got a lot of attention were [spread parameters](#), [destructuring](#) and [default parameters](#). Many developers found these features to be helpful for making their existing JavaScript code more powerful, concise and/or readable.

A few of the new features are also touted to [fix the “bad parts” of JavaScript](#). To add block scope and prevent variable hoisting outside of the scope, you can now use `let` and `const`. With [arrow functions](#), the variable “`this`” always points to the object that it is physically located within.

The [rest parameters](#) feature lets us treat the parameters like an array so that we can use the array functions like `slice`, `sort`, etc.

As developers get used to the new syntax changes and the benefits they yield, they may become more open to adding new features. Across the web, more and more tutorials are popping up that incorporate the ES2015 syntax without mention or explanation of ES2015. This suggests that it is becoming the new normal and that this trend will continue into 2017.

Classes: Objects and Prototypes

There were many discussions and strongly-held views about the inclusion of ES2015 Classes. Thankfully, people have made their points and the debate has seemingly to dying down.

As predicted, people stay steadfast in their preference to Object-Oriented Programming, Functional Programming, etc. The biggest point made about [ES6 Classes](#) was to specify that they were mostly syntactical sugar for prototypes and not to be

confused with traditional classes like those from Java.

Next year will not be the year that all developers decide they will all work together and only write functional JavaScript. As many developers have found first hand, simply getting five people to agree on semicolon usage is difficult enough. React had built a class system for its own framework but are very open to the [inclusion of ES2015 classes](#), as is [Angular 2](#). With frameworks making the use of classes easily accessible, it will be up to teams and developers to choose whether they want to be classy or classless in their code.

Additional Features

Last year, we suggested the proposals likely to be added to the language were Exponential Operator, `Array.prototype.includes`, SIMD.JS - SIMD APIs + polyfill and Async Functions. Let's see how far along these additional features are.

[Array.prototype.includes](#): This feature is still considered an ES2016 feature and is actually supported on Edge 14, Firefox 45+, Chrome 55+, Safari 10, Node 6.5+ and iOS 10.

[SIMD](#): This feature is listed as a “Candidate” at stage 3 but will be a large feature.

[Async Functions](#): This feature is the only “large” feature on the list for ES2017 features and are already supported by Firefox 52+ and chrome 55+.

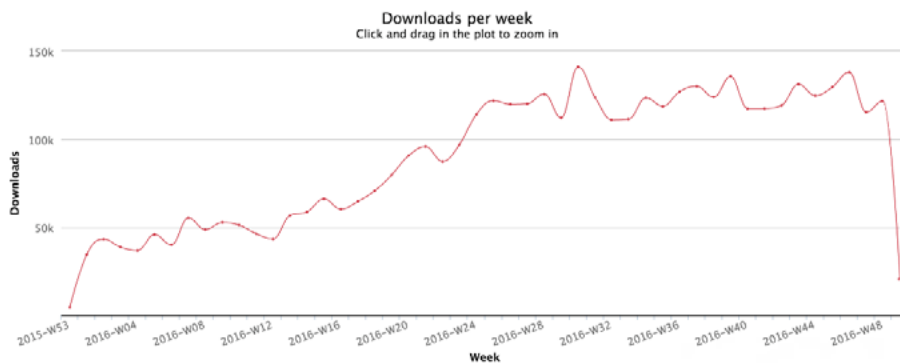
Package Managers

Going into 2016 we suggested using [systemJS](#)

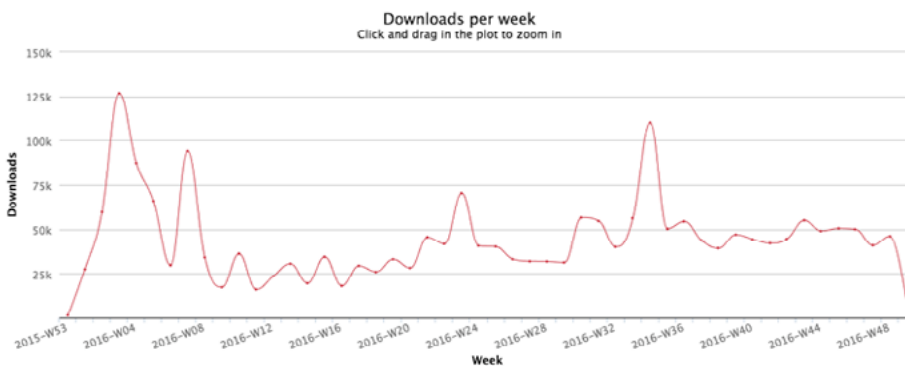
and jspm.io and it is still a solid option. In 2016 alone systemJS has been downloaded over 4.4 million times, ~520,000 times in the past month.

It seemed that people were leaning toward making npm the go-to package manager for the both front and backend. This still seems to be the case, especially with npm being paired with webpack and browserify.

The npm registry is still at the top, after all it provides access to over 300,000 packages and there are more than 4 million people using the registry. There is an advantage to using the same package manager if you are using Node.js as your backend.



systemJS weekly npm downloads from January 2016 to December from <https://npm-stat.com/>
jspm is coming in at ~2.2 million for the year and ~200,000 for the month.



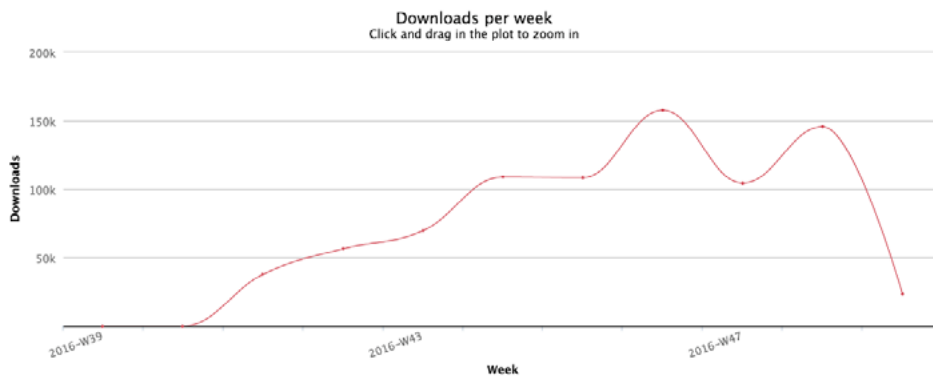
jspm's weekly npm downloads from January 2016 to December from (courtesy https://npm-stat.com)

Looking back on last year's predictions, we did not take into account the option of a new package manager coming onto the scene. Yet, that's exactly what happened when Facebook introduced Yarn in October.

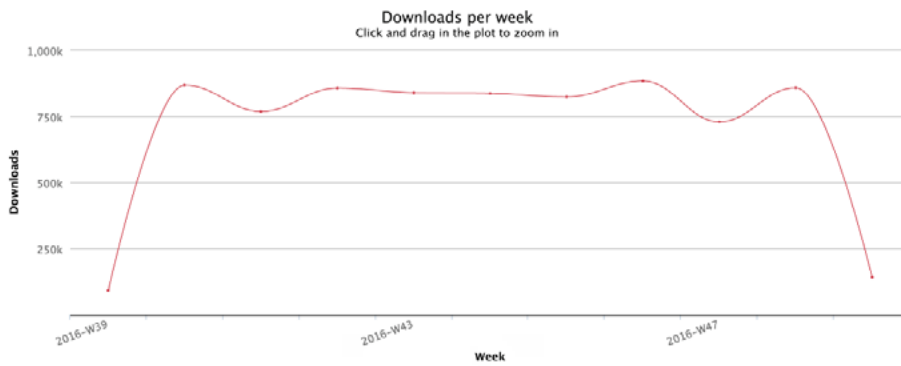
Facebook had been using npm but once their codebase and engineering team grew, the company started running into problems and decided to create its own package manager. Yarn piqued the interest of developers at launch thanks to the big names involved: Google and Facebook. Based on the npm download stats, Yarn took a dip while the U.S. was on Thanksgiving break and hasn't surpassed its ~160,000 weekly download record from November.



We'll have to keep an eye on Yarn but so far its November downloads (~538,000) aren't even coming close to npm's (~3.6 million).



Yarn's weekly npm downloads from October to December (courtesy <https://npm-stat.com>)



To be clear, Yarn is not a replacement for npm. It is a CLI client that fetches the modules from the npm registry. It's unlikely that we would not see another package manager surfaces in 2017 but Facebook may have opened the

Tool	Upvotes	Downvotes
npm + Browserify	51	3
Yarn	11	0
Bower	30	5

Pros + Cons

- npm + Browserify:**
 - PRO** Best way to share code with the backend. If you're using node.js as your backend, you gain a lot of flexibility by using the same package manager for the
 - PRO** Huge active ecosystem. Npm gains a lot from its large community, and the activity from node.js gives npm the largest set of active
- Yarn:**
 - PRO** The same results will be yielded every time yarn is run in a repository. One of the most important aspects of Yarn is determinism (predictability). The lock file ensures that
 - PRO** Can tell you why a package was installed. yarn why <query> can tell you why a package was installed and what other packages depend on it.
- Bower:**
 - CON** Does not store components in a registry. Bower installs components directly from uris and repositories, which makes it more susceptible to
 - PRO** Manages non-JavaScript components. Bower is flexible enough that you can manage pretty much any package you would need on the front-end, so

A snippet of the Slant survey results

field up to new contenders. Even with snafus like the [left-pad](#) situation, npm feels very reliable.

Slant.co created a [survey](#) for their users to be able to rate and explain the pros and cons of different front-end package managers.

So far, the top three products are npm and Browserify with a total of 51 upvotes and 3 downvotes, Bower with 30 upvotes and 5 downvotes and JSPM with 15 upvotes and 2 downvotes. It's important to note that there are only 169 votes, so this is a small pool of developers. Still, it's worth nothing the feedback and we should learn more over time.

Issues Resolved in 2016

We touched on a few key issues that we believed needed to be resolved in 2016, here's the update on each of those.

- “How native modules are loaded in a browser will need to be ironed out and an initial implementation will need to commence.”

As we touched on a bit before, this is still a work in progress. The good news is that this is something the technical committee is working on. The implementation has not happened but it seems to be on track for 2017.

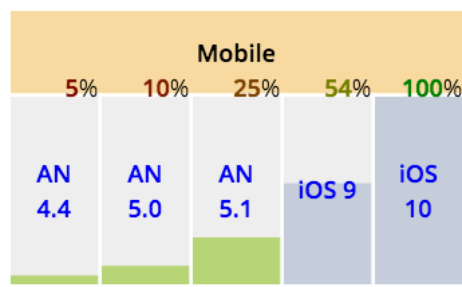
- “We haven't fully scratched the async itch. While, await functions will help, the journey is far from complete. Promises and eventually streams will need to be used throughout (e.g. HTTP promises). And O'yeah, canceling a promise. That might be a good idea.”

Promises are supported on all major browsers and async functions are set to be a 2017 feature.

- “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 TC39 are considering multithreading but it is hard to say when that will start making its way through the pipeline. There is a [proposal for parallelism with web workers](#), running scripts in background threads. It is a very complicated issue to work through but it would increase performance using multicore processors.

- “The ‘should we or shouldn't we’ debate about immutable native objects will hopefully conclude.”



The results for ES2015 compatibility on mobile.

The conversation around this hasn't been as active lately, presently but it seems to boil down to what programming paradigm you subscribe to. If you are truly using functional programming, you never attempt to mutate state. Therefore, it should not matter if the state is technically mutable. If you are using object-oriented programming, immutability is an odd fit because immutability is technically about data-structures. That being said, there are ways to make it work, if you're up for it. Here is a [thorough article](#) discussing just that.

- “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.”

Unfortunately, looking back at the ECMA Compatibility Table, it seems no one has been paid off. iOS has 100% compatibility but Android seems to be falling behind at a mere 25% for Android 5.1. With Google's recent push for [Progressive Web Apps](#) and focus on cell phone and tablet usage, all mobile browsers will have to catch up fast. Maybe then, the only ones getting a payoff will be us, the users!

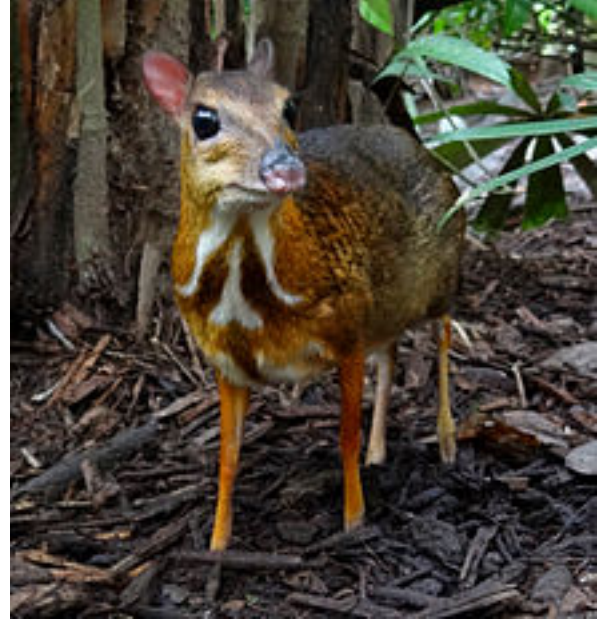
Options Overload

At the beginning of 2016, developers were already aware that the many different styles for constructing JavaScript applications were overwhelming. We had hoped that the developer community would be able to update the way we think about and teach JavaScript development to accommodate the variations. As the authors of this whitepaper, we do believe that the focus on best practices is present. Unfortunately, there are still so many ways to build JavaScript applications that it is hard to find multiple tutorials or examples that have the same application setup.

Although many developers seem to be suffering from JavaScript fatigue, the bigger issue may actually be from the [paradox of choice](#). In local JavaScript communities across the web there are always conversations about the pros and cons of using different JavaScript technologies and the best ways to implement new features. How do you commit when things are changing so fast?

New tools are popping up just as fast as old ones are dying. You can spend hours researching how to create your new JavaScript application. Then, after the onslaught of options, you end up paralyzed with indecision and instead decide you would be happier searching for [Chevrotains](#) in the rainforests of West Africa—which feels much more stable because they haven't changed for over 5 million years.

The good news is that developers, as a community, are becoming more aware of this problem. While we may not all decide to use the same style for how



How could you not want to search for these amazing creatures?!

we build our JavaScript applications but, we may hopefully slow the creation of new options. This is a very optimistic prediction but in 2017 developers may just begin to streamline approach to building applications.

Usage

Last year we mentioned WebAssembly stealing some of the spotlight from JavaScript once it hit all the browsers. It hasn't taken over the internet yet but the advancement of WebAssembly this year has been significant. V8 has a [WebAssembly Browser Preview](#) and the [WebAssembly Community Group](#) has their MVP and JavaScript API implemented on several browsers. They are planning for the Browser Preview to finish in Q1 2017, so we'll see what comes next very soon! If you want to get a taste, you can [check out the demo](#) using Chrome Canary and Firefox Nightly (you will have to switch some flags).



WEBASSEMBLY

It looks like a great year for WebAssembly

We did predict that JavaScript would become the language of native applications (NativeScript, Electron, React Native) because developers would want to write in JavaScript alone. The [State of JS](#) survey results from [Sacha Greif](#) confirm that developers may be easing their way into JavaScript mobile frameworks. As for desktop applications, Electron has reached over 1.6 million downloads and React Native follows close behind with 1.5 million since their releases in 2014 (over 200,000 and 180,000 in the last month, respectively). With this increase in downloads and the NativeScript plans to include desktop support, JavaScript is definitely infiltrating the native application scene and will continue to do so into 2017.

Nothing seems to be hindering the continued growth of JavaScript usage and with it being a language you can use for practically everything—(mobile, IoT (internet of things), native, back-end, front-end—more developers may be switching over. The only perceivable downfall I see is that the onboarding process for new users may be quite overwhelming. With new features coming out from ES2015 and ES2016+, just writing

JavaScript has many options. Once users decide on that, they then need to make a decision on frameworks, transpilers, package managers, modules and more. Nonetheless, all the powerful ways you can use JavaScript outweigh this issue.

JavaScript Wrap-Up

There are still a lot of exciting features that are going to be worked on and added in 2017. Our predictions from last year turned out to be pretty close but the hopes for better mobile browser and native module support will have to wait a little while longer. We'll check back next year to see if there are new package management tools, if we're over JavaScript fatigue and if everyone is referring to ECMAScript editions correctly. Until then, I think that we'll all be able to make great projects with JavaScript and continue to learn a lot while doing so.

2016 was a pivotal year for JavaScript developers. That seems rather ironic considering that every year is somewhat of a pivotal year since JavaScript and the web platform seem to be in a state of constant evolution. The best practices of yesterday are today's anti-patterns; yesterday's libraries, today's technical debt.

This makes it all together frustrating to feel like one has ever really “mastered” JavaScript and has even garnered a catch word of its own in the industry known as “[JavaScript Fatigue](#)”.

While change is inevitable and moving forward is always the best path, it is worth revisiting the past so that we can learn from it. It's in that spirit that we look back on how JavaScript evolved in 2016, and what its trajectory is for 2017; so that we can ready ourselves for the next big changes for JavaScript.

Libraries And Frameworks

It's no longer debatable that JavaScript has amassed a popularity that is unmatched in the software development world. This manifests itself largely by the sheer number of open-source frameworks that are released each year for JavaScript developers. The site [javascripting.com](#) attempts to catalog each of these different frameworks and their popularity—there are 73 pages of libraries available in total.

While there are innumerable JavaScript libraries for various pieces of functionality (User Interface, Date Parsing, Data Storage, etc.), developers will be primarily familiar with the so-called JavaScript Frameworks; those libraries with the purpose of helping you compose the different pieces of your application.

In the [State of JavaScript Survey from Sasha Greif](#), the libraries that made the cut for awareness were React, Angular 2, Ember, Vue and Backbone. In addition, Aurelia gets a nod on the strength of its repeated

occurrence as a write-in candidate. For those reasons, we're going to look at each of these frameworks, substituting Aurelia for Backbone given the age of Backbone and its likely appearance as a legacy. Looking back can help us determine how each of these frameworks impacted web development in 2016, as well as where they are likely headed.

First, it's important to recognize the fundamental way in which the model of open-source has changed.



An Open-Source Shift

JavaScript has long enjoyed an almost entirely open-source pedigree. Up until now that was primarily on the strength of some remarkable individuals, such as [John Resig](#) (jQuery), [Jeremy Ashkenas](#) (Backbone, Underscore), [Thomas Fuchs](#) (Zepto, [script.aculo.us](#)), [Mihai Baizon](#) (Uglify), [Eric Schoffstall](#) (Gulp), [Ben Alman](#) (Grunt), and a host of others. The community would rally around these projects and remarkable things were accomplished. The entire web was run primarily on the work of hundreds of individuals who had never even met.

Angular was the first open-source library to change this landscape. The Angular project is primarily built and controlled by Google. There is a team of developers, marketers and the like that are paid by Google to work on this project full time.

React became the second entry in this category of open-source. Initially created at Facebook, it is heavily promoted and marketed by Facebook, which also pays a team of developers to work on React and React based tools and frameworks, such as React Native.

Despite this, both frameworks are truly open-source in the classical sense that they have enormous communities that surround and contribute to their success. The defining difference is that at the end of the day, large corporations own and make the ultimate calls on these projects.

Open-Source Predictions for 2017

Given the now-large corporate involvement in open source, it's likely this trend continue to grow more prominent in 2017. Look for players such as Microsoft or even Apple to join the fray with their own large open-source offerings for JavaScript developers.

Angular 2

We opened last year's discussion of frameworks with React, but 2017 likely belongs to Angular 2, so we'll start there.



Last year, we predicted that Angular 2 would be released in the first quarter of 2016.

A release candidate was announced in May at ng-conf, but there ended up being five release candidates and each was a large breaking change from the previous, which did continue the instability of Angular through to the middle of September, when Angular 2 Final was released.

In addition to the core framework, the Angular team also released a [command line interface](#) tool to help control the complexity of an Angular 2 applications and scaffolding out commonly used boilerplate.

Angular Predictions for 2017

Given the amount of interest in Angular—despite it's rough road to final release and dozens of breaking changes—it's clear that Angular enjoys a level of trust and adoption that virtually guarantee that Angular 2 will be the dominant framework of 2017.

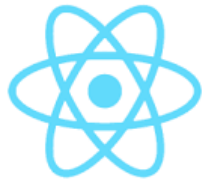


“What does ‘final’ mean? Stability that’s been validated across a wide range of use cases, and a framework that’s been optimized for developer productivity, small payload size, and performance. With ahead-of-time compilation and built-in lazy-loading, we’ve made sure that you can deploy the fastest, smallest applications across the browser, desktop, and mobile environments. This release also represents huge improvements to developer productivity with the Angular CLI and styleguide.”

Jules Kremer – Angular Team

Angular 2 has some concepts that make it remarkably unique, including module theory and its complete decoupling from the DOM. This makes frameworks such as [NativeScript](#) possible so that developers can build native mobile apps with the same knowledge that they use to build web applications.

The end of 2017 will likely see the release of Angular 3.0. While the team describes this as simply semantic versioning, version 3.0 will present an opportunity for the team to introduce necessary breaking changes. That said, we will not see the API change drastically from what it is in 2.0.



React

React

React was an anomaly in 2015, and that trend continued with force in 2016. React is only a portion of the full front-end framework solution that most developers are looking for, which is the major difference between it and the other frameworks discussed here. That makes it very hard to draw a direct comparison.

In 2016, we predicted that React's popularity would continue to grow, especially with consumer-facing applications. This turned out to be extremely accurate. While companies are slower to adopt React for enterprise-facing applications, React is seeing a lot of use for consumer applications, with big names such as Airbnb, Dropbox, eBay, Expedia and even internet behemoths such as Netflix using React.

We predicted that there would be continued controversy around JSX, which is the way React mixes HTML in JavaScript in an XML-like fashion. However, this melted into a complete nonissue in 2016, with nobody even batting an eye at this concept anymore.

We predicted that 2016 would be the year of the commercial React ecosystem. This turned out to be incorrect. While the open-source community is quite large, it is still very difficult to find complex, commercial-grade React components from well-known vendors.

We predicted that enterprises would continue to watch React from a distance. This turned out to be largely true with the RC and Final releases of Angular 2 completely overshadowing React in the enterprise. This list of companies that use React [confirms this assumption](#).

React Predictions for 2017

Considering that React does the few things that it does so well, it's not likely that we will see a new or different version of React in 2017.

Given that Facebook weighed in on the React Starter Kit landscape by releasing the "Create React App" package, it's likely that we may see the social media giant release other official React components. It's easy to speculate that the React Router project may be merged into the official React repo at some point.

It's also somewhat likely that React will release its own UI component framework in 2017. This is because Facebook itself has a lot of standard UI and CSS components. Given the [recent trend to package and open-source virtually everything the company does](#), we would not be at all surprised to see a Facebook Bootstrap of sorts.



Vue

Vue didn't even make our cut last year when we made framework predictions, and that's because it simply wasn't on the radar at that time. Since then, Vue has garnered a decent amount of attention from the JavaScript community.

As of the time of this writing, Vue is trending on GitHub with 122 stars just today and over 35K all time. Compare that with [Ember which has 17K stars](#) and [Angular with 53K](#). There is no denying that Vue is a contender.

Vue is different from all the other players in this whitepaper primarily due to its simplicity. Vue is likely the easiest of the modern JavaScript frameworks to work with. Its API is similar in some ways to Backbone (such as specifying elements and data for chunks of HTML) and there is also some influence from Angular in terms of using special custom HTML attributes to easily bind the DOM to Vue models. It also doesn't eschew classic web development the way React does with JSX. Its single script include is a breath of fresh air in an era of JavaScript build systems that tend to cripple developers with their complexity.

Vue Predictions for 2017

Due to the intentional simplicity of Vue, its grass-roots success and the constant draw of web developers back to the core concepts of the browser, we predict that Vue will unseat React in 2017 as the light-weight front-end framework of choice for consumer facing applications.

That may seem like quite the statement, and is probably the wildest prediction of this whitepaper, but Vue contains all the elements of projects past that have taken the web by storm (see Bootstrap, jQuery), and unlike React and Angular, it is not built by a for-profit corporation, which is more true to the basic tenants of the open web.

Enterprises will continue to favor Angular due to its strong corporate backing element.



Ember

In 2016, we didn't say much about Ember, other than that it was a "sleeper" framework and would continue to be just that. This is largely the case. Ember has a loyal cult following, but it tends to be like a musical act that everyone has heard of, but few people listen to. However, those that do will swear that it's the best show of all time.

We predicted that Ember would be the popular alternative to React for those consumer-facing applications, but it appears that honor has gone to Vue.

It should be noted that it is technically possible to use React alongside something like Ember. This is because React only solves part of the full stack JavaScript problem—specifically the view part. That means that it can also be used with Angular, although we typically do not see developers mixing React with another large-ish framework—Flux and Redux notwithstanding.

Ember Predictions for 2017

We don't have any predictions for Ember in 2017. Much like jQuery and Backbone, this is a framework that is mature and unapologetic in its implementation. The only prediction one could safely make is that none of this will change.

Aurelia made our list last year and we had several predictions for the somewhat niche front-end framework. Aurelia frequently shows up in requests from customers that we talk to, and it shows up more often than any other framework in this whitepaper besides Angular.



What is it about Aurelia that developers seem to love so much? It could be the fact that it comes from the creator of [Caliburn.Micro](#), which enjoyed massive success inside of the .NET community. It could also be because it relies almost entirely on just plain JavaScript constructs and doesn't involve a lot of boilerplate. Whatever the reason, Aurelia has won the hearts and minds of some section of developers and deserves a look from anyone looking for their next JavaScript framework.

In 2016, we predicted that developers would adopt Aurelia in droves over the course of the year. While Aurelia seemed to hang on to its dedicated core, we did not see strong increase in the interest of Aurelia over 2015. The Google search trends show roughly the same sentiment over 2016.



We also predicted that Aurelia might see a native counterpart, such as NativeScript or ReactNative. This also did not turn out to be true, despite Aurelia's explicit goal to be more than just a web framework.

We predicted that large enterprises would begin to adopt Aurelia since it was an officially supported product. This also turned out to be largely incorrect as Angular continues to dominate the enterprise JavaScript spectrum.

[Rob Eisenberg](#) recently published [an article on the future](#) of Aurelia which makes it much easier for us to speculate on the future. Of note from his article was the intention to create UI components specifically for Aurelia.



“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.”

Rob Eisenberg, Creator of Aurelia

Aurelia Predictions for 2017

Aurelia is a fascinating alternative to Angular and React, and we're continually inspired by the work that Rob and his team do on the project. However, the sheer dominance of Angular 2 and React (or Vue) leave little room for anyone else besides niche players. While not much of a prediction, our guess is that this will remain the case for Aurelia in 2016. We also think that it will likely lose developers to Angular 2, which shares some concepts with Aurelia, such as using plain JavaScript classes as the binding context.



While not a front-end framework like the rest of the items in this list, [Progress® Kendo UI®](#) is a bit of an anomaly. It is first and foremost a UI library of widgets and components. However, the version of [Kendo UI based on jQuery](#) does contain portions of full stack framework features, such as two-way binding, routing and view management. This qualifies it for inclusion in the whitepaper. Aside from that, Progress makes it so we know a little about its future.

In 2016, Progress launched [Kendo UI for Angular 2](#) Beta, which was a complete rewrite of Kendo UI to use Angular 2 as the underlying framework for DOM manipulation, binding, routing and the like. This enables Kendo UI to leverage all the advantages of Angular 2, such as:

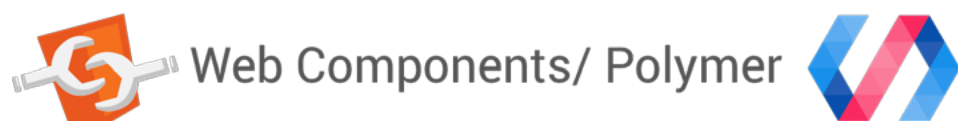
- Binding speed
- Ahead-of-time compilation
- Dependency management

Kendo UI Predictions for 2017

To be fair, Progress makes [Kendo UI](#) so we know a bit about where it's going. To that end, expect a full release of Kendo UI for Angular 2—which includes all the widgets in the Kendo UI portfolio—by May of 2017.

Progress will also continue to work on Kendo UI for jQuery in 2017, as the company doesn't see jQuery going anywhere and it's still the most popular way for customers to build their applications.

In addition to the UI framework itself, Progress will release Kendo UI Builder, which is a visual tool for designing user interfaces composed of Kendo UI components. While currently limited to [OpenEdge](#) data sources, a mature Kendo UI Builder would connect to any data source to enable easy drag-and-drop composition and configuration of user interfaces with real-time visual feedback.



Web Components / Polymer

We'd like to close out our section on frameworks by discussing what may be the most important technology that the web community has yet to adopt: Web Components. We've lumped Polymer in here as well because it is Google's polyfill library for Web Components which are largely unusable cross-browser without it.

Web Components are a standard for the way that developers build and deploy components for web applications. These are typically thought of as visual components, or rather custom HTML elements, but they can also include processes that occur in the background, such as AJAX. Web Components are so critical because they are the only thing that will be able to curtail the Cambrian explosion of JavaScript libraries, all of which may implement components in a different way and are usually only usable with a specific JavaScript framework.

Unfortunately, React inadvertently put the brakes on Web Components by creating a simple and elegant component model that worked across all browsers without polyfills or hacks. The rapid adoption of React meant that developers' interest in Web Components was negated by their frenzy for React, which offered a similar yet much slimmer solution. It also highlighted that web components in their current state are still not meeting developers where their needs are.

In 2016, we predicted that all major web browsers would support Web Components by the year's end. This is simply not the case. The following chart shows the state of browser support for Web Components as of the time of this writing.

	Spec'ed	Implementation				
		Polyfill	Chrome / Opera	Firefox	Safari	Edge
Templates	Yes	Yes	Stable	Stable	8	13
HTML Imports	Yes	Yes	Stable	On Hold	No Active Development	Vote
Custom Elements	Yes	Yes	Stable	Flag	Prototype	Vote
Shadow DOM	Yes	Yes	Stable	Flag	10	Vote

Basically, Chrome is still the only browser that fully supports Web Components. Firefox has put HTML imports on hold and Custom Elements and Shadow DOM are still behind flags. Safari has remained annoyingly silent on HTML Imports, but in a surprising pivot decided to ship an implementation of Shadow DOM. While Edge appears to be the holdout, they have announced intent to ship support for HTML Template elements. They have not, however, fully committed to Web Components, citing that they will instead ship support for features as they become stable pieces of the Web Components Standard.



“Following template support, and after completing the DOM rewrite, the next goal is to implement Shadow DOM, the second-hardest feature to polyfill, followed by Custom Elements. We plan to evaluate the rest of the first generation of Web Component specs after that. Naturally, as the specs continue to evolve and additional web component-related technologies rise in importance we may shuffle priorities along the way.”

Travis Leithead and Arron Eicholz – [Microsoft Edge and Web Components](#)

We also predicted that JavaScript frameworks would begin to swap out their own component implementations in favor of Web Components. Given that Web Components aren't fully baked, this has not happened.

However, Angular 2 was designed from the beginning to support Web Components. They even ship their own Shadow DOM emulation. In other words, when Web Components are ready, only Angular 2 is specifically designed to use them. This is another reason that we are building many of our own components on Angular 2s infrastructure, so that when Web Components are ready, our own leap won't be nearly as far.

JavaScript in 2017—Beyond the Browser

As the technology world has evolved, JavaScript has evolved with it. In previous years, that meant JavaScript's inclusion in software worlds it was never originally intended for, like server-side apps, mobile apps and robots. And today, JavaScript's growth has brought the language to chatbots, virtual reality, IoT and a whole lot more.

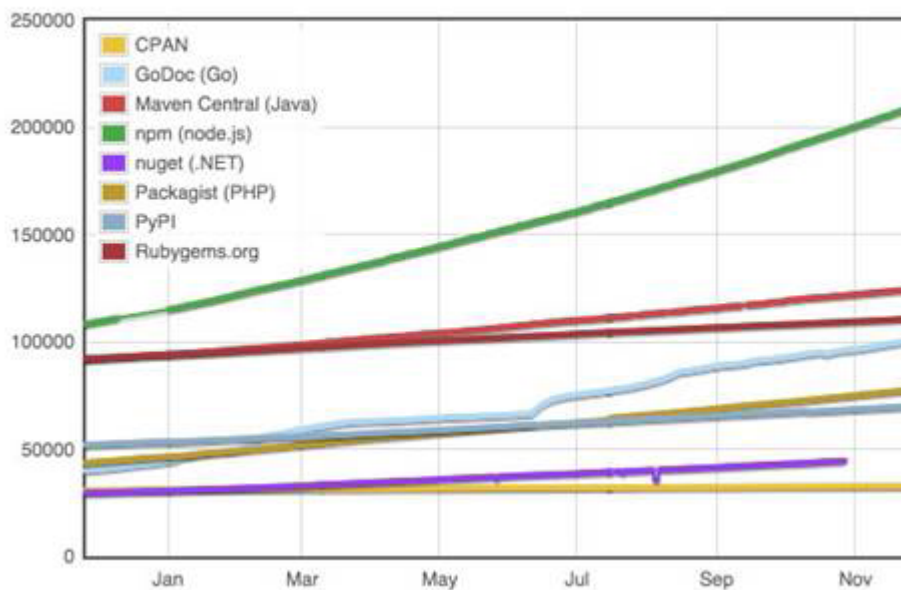
In addition to reaching new frontiers, JavaScript's role has become more established and stable in ecosystems in which it has long been a part of, such as server-side Node.js apps, as well as mobile and desktop application frameworks. In this whitepaper, we'll look back at some predictions we made a year ago for JavaScript in each of these software worlds, and then make some predictions about where JavaScript is heading outside of a browser in 2017. Let's start by looking how JavaScript is doing in server-side app development.

Node.js

Node.js is an open-source runtime library for building both server-side apps, as well as small bits of JavaScript code you need to run outside of a browser environment. In the past few years Node went from a niche technology popular in startups, to a mainstream development approach used by companies of all sizes.

Node's package manager, npm, has transformed from hosting utility modules for server-side apps to the canonical place to store distributable JavaScript code. Perhaps the best indication of Node's rise is the sheer number of packages stored on npm. In last year's predictions, we included the following chart to show npm's dominance over package managers in alternative languages.

Module Counts

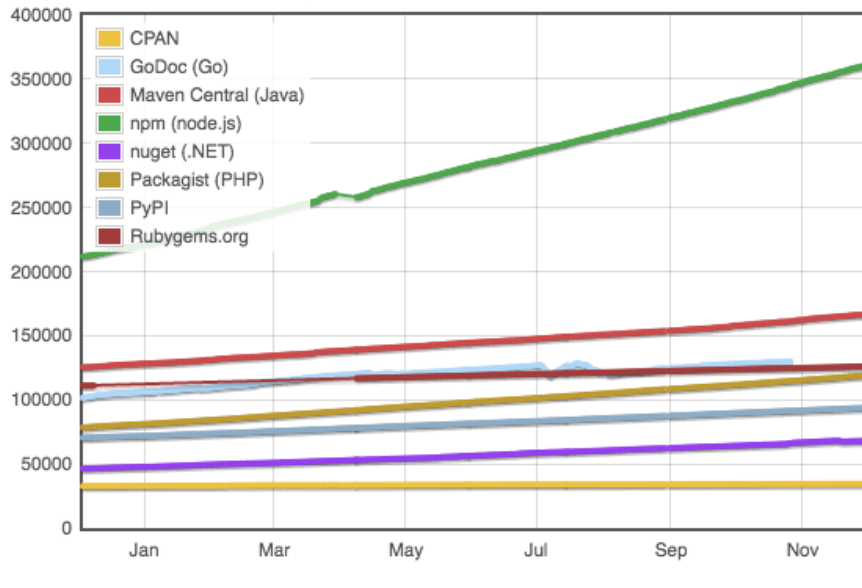


Module counts from modulecounts.com as of December 2015

Fast forward one year and npm's growth shows no signs of slowing down. In fact, npm's move from ~200,000 to ~350,000 packages has forced the Module Counts site to reconfigure its chart's Y axis.

There are a number of factors that have led to this increase, and one of them is a growing number of enterprise companies using Node in their infrastructure. In last year's discussion, we made the following prediction.

Module Counts



Module counts from modulecounts.com as of December 2016

“In 2016 expect to see further adoption of Node 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 adoption in the enterprise, replacing typical enterprise solutions like .NET and Java.”

This wasn't exactly a risky or unique prediction given Node's growth, but it seems to have been accurate. [Node's own case study page](#) has a small list of not-very-small companies that have now adopted Node, including the likes of Netflix, GoDaddy and Capital One.

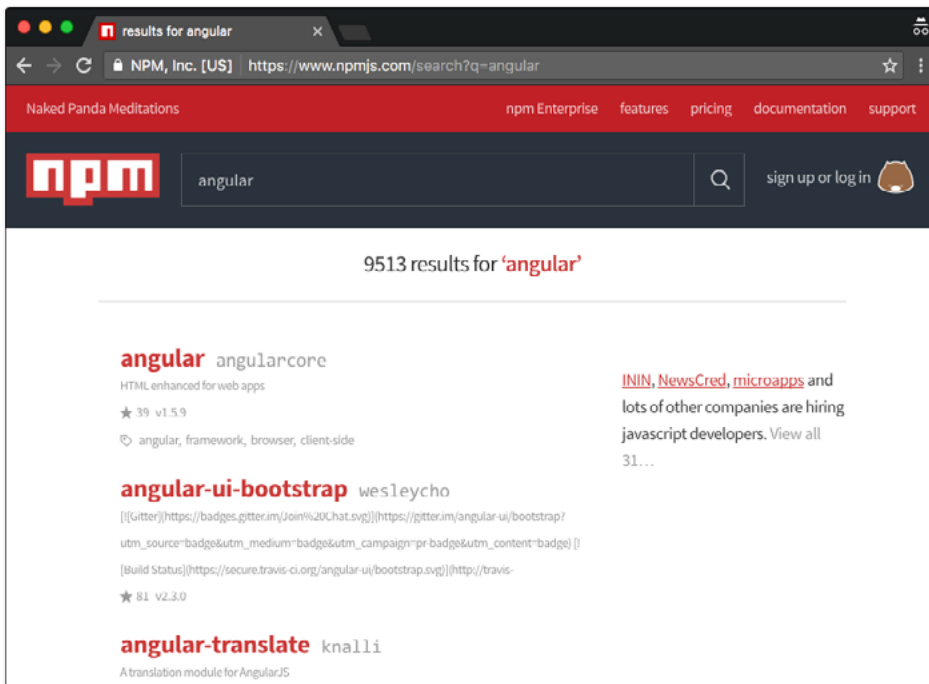
But perhaps the most telling sign of Node's use in critical infrastructure comes from the first company listed on that page—NASA. You can read [Node's case study on NASA](#) for yourself, but I'll drop in an excerpt here just to give you an idea.

“When you've got the safety of astronauts on the line, little hiccups and service interruptions turn into life-and-death situations. From EVA [extra vehicular activity] data to astronauts up in space, Node.js helps ensure there's a safe home for everything and everyone.”

But it's not just the NASAs of the world driving Node's growth. Node's package manager, npm, has become the de facto choice to store JavaScript code across all environments—and that consolidation on a single package manager helps drive adoption of Node itself.

Literally every framework and technology we discuss in this article use npm to store and distribute their source code. A quick npm search for “jquery”, “polymer”, “react”, “cordova”, or “nativescript” can give you an idea of the sheer scale that npm operates at now. As JavaScript grows in popularity, npm grows in popularity. And as npm grows in popularity, so does Node.js. And there's no reason to believe that this trend will end anytime soon.

Searching for “angular” on [npmjs.com](https://www.npmjs.com) returns nearly 10,000 results. Angular is one of many libraries that is distributed via npm.



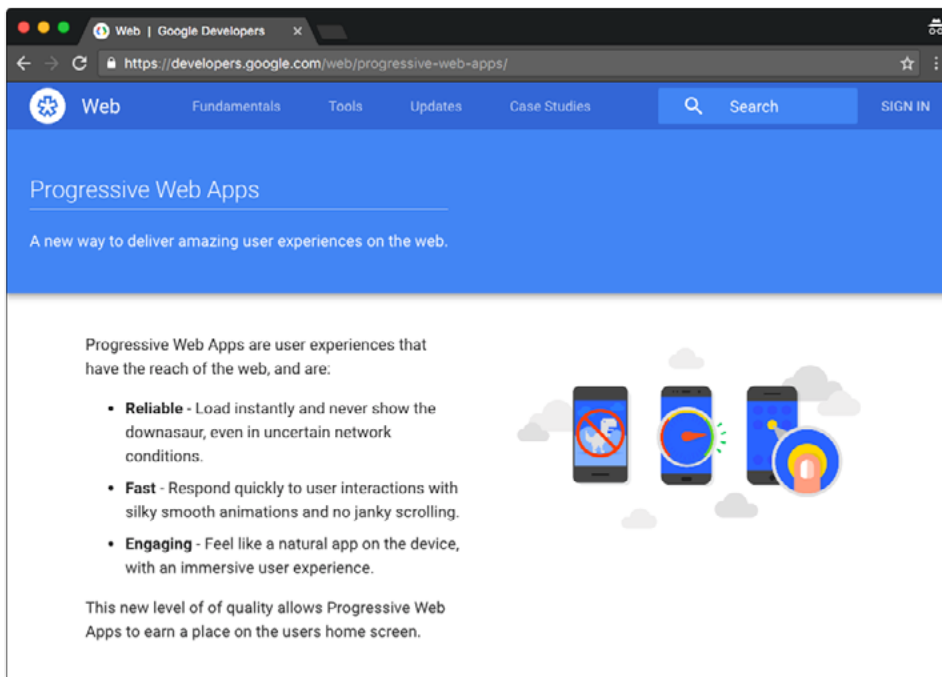
In 2017, we believe more companies will make the switch to Node from more traditional development approaches like Java and C#. We believe [TypeScript](#), a Microsoft-written superset of JavaScript will help drive Node's growth, as its features make JavaScript a more approachable language for Java and C# developers. Node's commitment to [long-term support releases](#) will also contribute to this growth, as it gives these companies a guarantee that the version they use will be supported in years to come.

Overall, large enterprises do not like maintaining multiple development systems and language, and Node enables these companies to consolidate on a single language for all of their development. And that consolidation applies to more than just server-side code. Let's take an updated look at how JavaScript is affecting the mobile world as well.

PhoneGap and Cordova

[PhoneGap](#) and [Cordova](#), the open-source framework on which PhoneGap is built, were JavaScript's first foray into the world of native mobile development. Cordova's basic approach is to wrap web code in a [WebView](#), and then use that WebView to drive a native mobile application. This approach enables web developers to build mobile apps with skills that they already have—namely JavaScript—and because of that, Cordova has remained a compelling option for building mobile apps for many years.

But that's starting to change. Today, Cordova is being challenged by alternative development approaches, most of which leverage the same JavaScript-based skill set that Cordova development is known for.

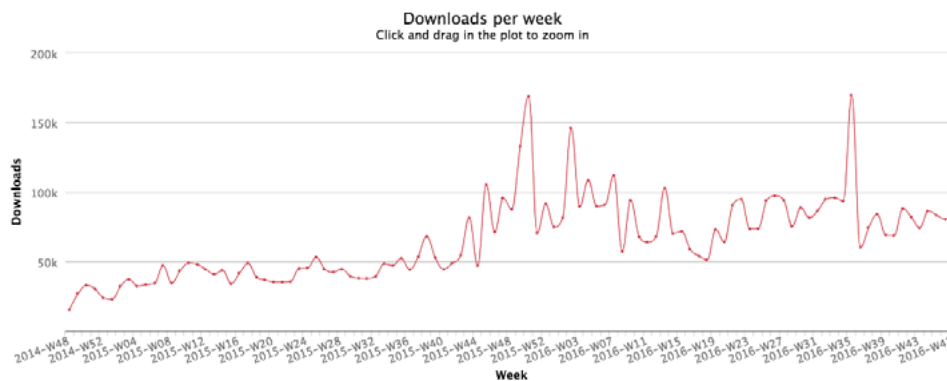


Google's home page for [Progressive Web Apps](#)

Perhaps Cordova's biggest challenger is the Google-led concept of Progressive Web Apps, or PWAs.

PWAs bring many native-like features to the web world, such as push notifications, offline access, and home screen icons. Last year, we predicted that Google would start pushing the PWA approach a little. That prediction has turned out to be, well, wrong—as Google has made it clear that the company is heavily committed to the PWA approach through a number of events. The recent [Chrome Developer Summit](#) featured a staggering number of talks on PWAs, as did this year's [Google I/O conference](#).

PWAs are relevant for our discussion because they eat into the primary use case of Cordova apps—web apps that need a bit of native functionality. If you have a web app that needs offline access or push notifications, building a PWA is a compelling alternative to building a Cordova-based native app. Although it's hard to gauge how many people are choosing PWAs over hybrid apps, most data shows that Cordova usage has flatlined or is declining. For instance, here are [Cordova's weekly download numbers for the last two years](#). As you can see, although Cordova's numbers are still very healthy, the trend line is no longer heading upwards as it was this time last year.



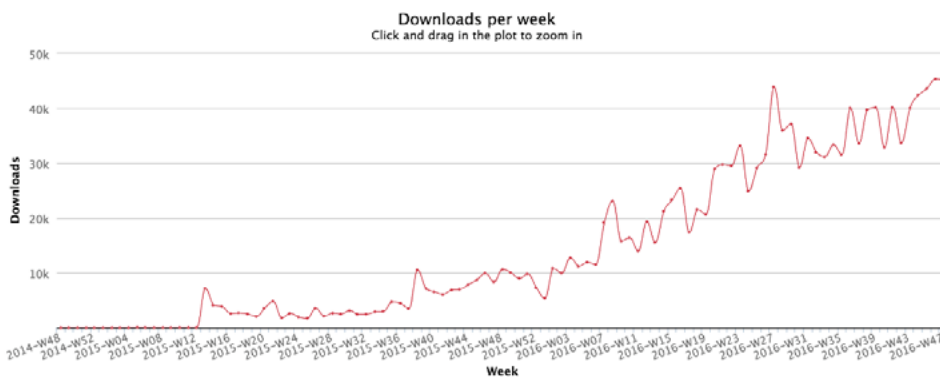
Weekly downloads of the "cordova" npm package from December 2014 until December 2016. Data from [npm-stat.com](#)

But there's another factor playing into this decline. Although we believe PWA usage is eating into Cordova's usage, we also believe a relatively new entry in the mobile world is taking market share from Cordova as well.

Native Mobile Apps

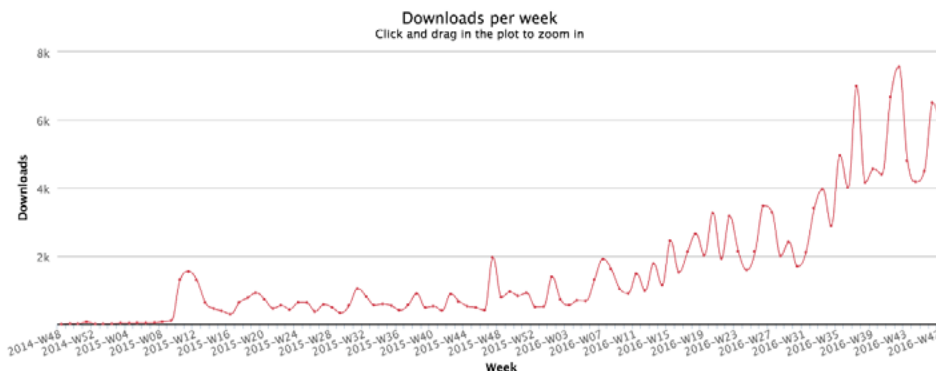
Pioneered by Appcelerator, the concept of a JavaScript-driven native app was popularized by a few new entries in the space—namely, Facebook’s React Native and Progress’s NativeScript. JavaScript-driven native apps do not use a WebView, therefore, they don’t suffer the same web-based performance problems that can plague Cordova-based applications.

In last year’s discussion we predicted that 2016 would be a year where these frameworks matured and started to see widespread usage, and that prediction appears to have been accurate. For example, you can see a continuous increase in [React Native’s weekly download numbers over the last two years](#).



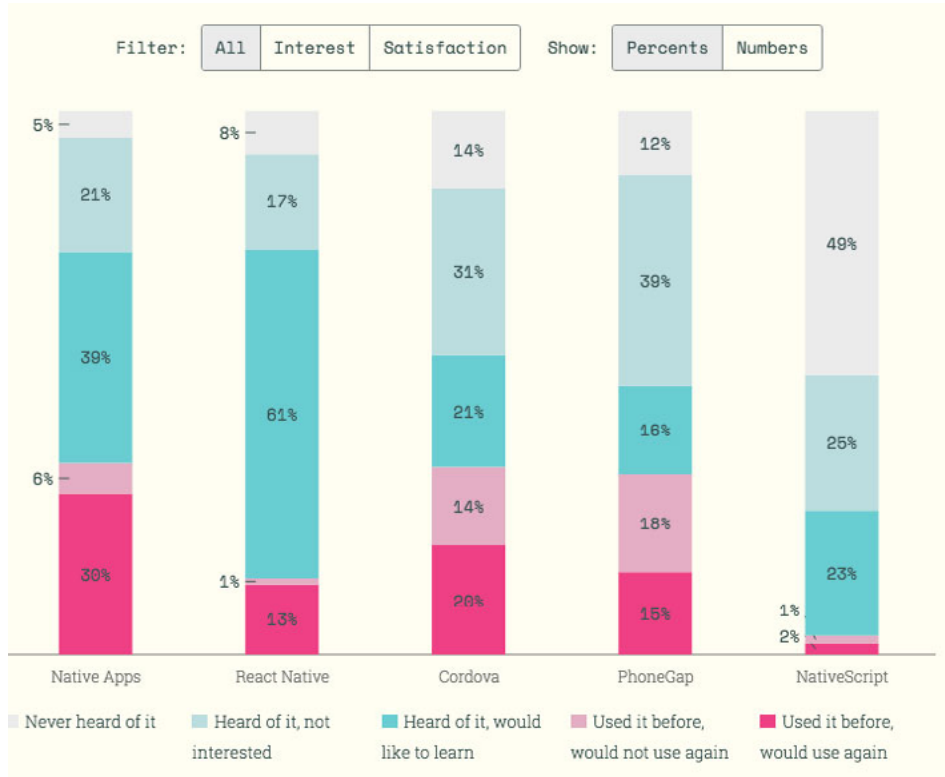
Weekly downloads of the “react-native” npm package from December 2014 until December 2016. Data from [npm-stat.com](#)

The [same trend line is also present for NativeScript](#).



Weekly downloads of the “nativescript” npm package from December 2014 until December 2016. Data from [npm-stat.com](#)

And it's not just download numbers that are up for these JavaScript-driven native frameworks. The recent [State of JavaScript 2016 survey](#) shows that JavaScript developers have a lot of interest in React Native, as well as burgeoning interest in NativeScript.



Survey results from the State of JavaScript 2016 survey on interest in mobile development approaches

The analysis of the State of JavaScript survey sums up these results quite well.

“Cordova and PhoneGap (which are basically the same thing) have much lower interest ratings, and it makes you wonder if people are turned off by the performance issues you sometimes hear about. With Cordova and PhoneGap, you rely on the underlying phone browser and its JavaScript engine to do the heavy lifting, which is often slower than running native code like React Native.”

In 2017, we expect the growth of these JavaScript-driven native frameworks to accelerate, as more and more JavaScript developers look to build mobile apps. React Native stands to gain from the [continued enormous usage of the React framework](#), and NativeScript—which [announced complete Angular 2 support in May](#)—stands to gain from the growing number of developers upgrading from Angular 1 to Angular 2. We also expect JavaScript-driven native frameworks to attract native iOS and Android developers, as JavaScript-driven native frameworks allow you to build truly native apps from a single codebase—not two.

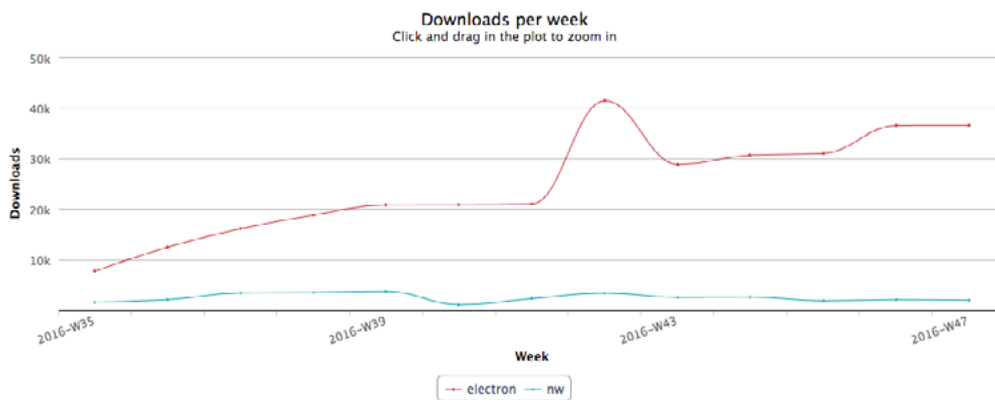
On mobile JavaScript is increasingly encroaching on territory that was once dominated by languages like Objective-C and Java. But that’s not the only new territory where JavaScript is gaining usage; let’s move our discussion to the topic of desktop applications.

Desktop Apps

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

In last year’s discussion, we talked about [NW.js](#) and [GitHub’s Electron](#), the two most popular JavaScript frameworks for building desktop apps, and theorized that each of their usage would increase dramatically in 2016. In reality, that growth has occurred—but only for Electron, which has established itself as the de facto choice for JavaScript-based desktop application development.

For example, if you compare [npm downloads for the “electron” and “nw” JavaScript packages](#), you’ll see that Electron (the red line) is now operating at a scale that competes with the likes of React Native, while NW.js downloads are relatively flat.



Weekly downloads of the “electron” and “nw” npm packages from September 2016 to November 2016. Data from [npm-stat.com](#)

In December of 2015, [Electron had 20,000 GitHub stars](#) and NW.js had 25,000; today, Electron has nearly 40,000 stars while [NW.js has just over 30,000](#).

Electron has also started to gain traction for mainstream desktop apps. The framework now powers the [Visual Studio Code](#), the popular editor from Microsoft that [boasted over half a million users back in April](#). Electron has also managed to perform the rare act of gaining popularity in both the React and Angular communities, and it's easy to find tutorials for Electron usage with both frameworks on the web.

In 2017 we expect Electron's dominance to continue. We expect to see further Electron tooling integration with the web's most popular frameworks—mostly React and Angular—as well as increased attention from software vendors. And as JavaScript continues to break into worlds traditionally dominated by Java- and Microsoft-based technologies, we expect Electron to continue to be used as an alternative to approaches such as WPF, Java, Adobe Air.

The appeal of using a single language for all your development needs is strong, and it's even taking JavaScript to some of the hippest and newest development approaches out there. Let's end our discussion with a look at JavaScript in a handful of brave new software worlds.

JavaScript's New Frontiers

If you ask analysts about what's coming in the development world, you'll likely hear a lot buzzwords like virtual reality, chatbots and IoT.

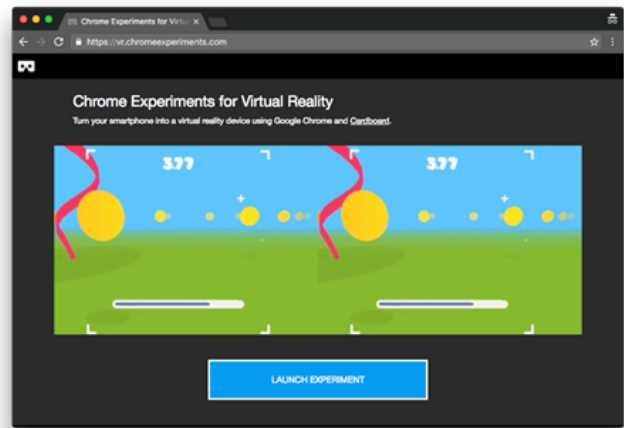
Of all of these new technologies JavaScript is biggest in the chatbot ecosystem, where people are using JavaScript to build everything from simple Slack bots to [far more complex bots used for tasks like commerce transactions](#). Most frameworks in the chatbot world include Node libraries in their SDKs, such as [Botkit](#), [Microsoft's Bot Framework](#), and [Facebook's wit.ai](#). [Microsoft's Bot Framework's documentation](#) includes the following quote on why you should build bots with Node.

“Bot Builder for Node.js is a powerful framework for constructing bots that can handle both freeform interactions and more guided ones where the possibilities are explicitly shown to the user. It is easy to use and models frameworks like Express & Restify to provide developers with a familiar way to write Bots.”

The same desire to reuse JavaScript skills has led many popular IoT libraries such as [Losant](#) and [zetta](#) to offer Node APIs, as well devices such as the [Leap Motion](#). There's also nascent interest in using JavaScript in virtual reality environments, led by the Google Chrome team as well as the [A-Frame framework](#).

That being said, JavaScript is still a niche player in many of these fields, with more performant players such as C++, Python, and C# retaining a dominant role. For example, the popular Oculus Rift device largely uses C++, and Microsoft's HoloLens requires you to write in C#.

We expect this trend to begin to change in 2017. As JavaScript gains in popularity, and as JavaScript's speed continues to increase, the language will continue to find inroads into environments like VR and the IoT. And as new software development ecosystems pop up we expect JavaScript to increasingly be included as a first-class citizen.



The Google Chrome team maintains an impressive set of JavaScript-built virtual reality experiments that you can [try for yourself](#).

JavaScript For All The Things

Ten years ago, using JavaScript on the server was unthinkable. Today Node has [3.5 million users and an annual growth rate of 100%](#). Five years ago, using JavaScript to drive a native iOS or Android app was a niche; today NativeScript and React Native are growing at staggering rates. Three years ago using JavaScript to build desktop apps was rare; today [Electron is downloaded over 100,00 times each month](#).

JavaScript will never be used for all programming, as many other languages are better suited to solve certain problems and use cases. However, JavaScript's widespread usage ensures that it will always be a factor, regardless of the development platform. Perhaps [Jeff Atwood's famous quote on the topic](#) is the best way to wrap up this discussion, as his statement has never seemed more prophetic.



“[A]ny application that can be written in JavaScript, will eventually be written in JavaScript.”

Jeff Atwood

Brought to You by Progress® Kendo UI®




Kendo UI delivers everything you need to build modern web applications under tight deadlines. Choose from more than 70 UI components and easily combine them to create beautiful and responsive apps, while speeding development time by up to 50 percent.

Try Kendo UI

About Progress

Progress (NASDAQ: PRGS) offers the leading platform for developing and deploying mission-critical business applications. Progress empowers enterprises and ISVs to build and deliver cognitive-first applications, that harness big data to derive business insights and competitive advantage. Progress offers leading technologies for easily building powerful user interfaces across any type of device, a reliable, scalable and secure backend platform to deploy modern applications, leading data connectivity to all sources, and award-winning predictive analytics that brings the power of machine learning to any organization. Over 1700 independent software vendors, 80,000 enterprise customers, and 2 million developers rely on Progress to power their applications. Learn about Progress at www.progress.com +1-800-477-6473.

Worldwide Headquarters

Progress, 14 Oak Park, Bedford, MA 01730 USA
Tel: +1 781 280-4000 Fax: +1 781 280-4095
On the Web at: www.progress.com
Find us on  facebook.com/progresssw
 twitter.com/progresssw
 youtube.com/progresssw
For regional international office locations and contact information, please go to www.progress.com/worldwide

Progress and Kendo UI are trademarks or registered trademarks of Progress Software Corporation and/or one of its subsidiaries or affiliates in the U.S. and/or other countries. Any other trademarks contained herein are the property of their respective owners.

© 2017 Progress Software Corporation and/or its subsidiaries or affiliates. All rights reserved.

Rev 01/2017 | 161214-0019

