01/31/2019 • 9 minutes to read

In this article

August 2016

Volume 31 Number 8

[The Working Programmer]

How To Be MEAN: Exploring ECMAScript

By Ted Neward | August 2016

Welcome back, MEANers.

We're in the middle of 2016. What you might not realize is that JavaScript (which is actually officially known as ECMAScript) has a new version of the language, ECMAScript 2015. If your JavaScript code isn't starting to use it, then it's high time to start. Fortunately, it's trivial to do so, because there are good reasons to do so. In this installment, I'll go over some of the more interesting and important features of ECMAScript 2015, and in a future piece, I'll look at what using MEAN in the more modern style would be like (essentially, now that you've got a firm footing on how MEAN applications operate, I'll reboot the code base completely; more on that then).

Simple Changes

Some of the simplest changes to the language simply embrace modes of programming that have been widespread convention in the ecosystem and community for some time now.

One of these conventions is that of immutability, which ECMAScript 2015 embodies through use of a const declaration:

JavaScript Copy

```javascript
const firstName = "Ted";
```

This, like the C++ declaration of the same name, declares that the reference name ("firstName") can never point anywhere else; it doesn't insist that the object to which the reference points cannot change. ECMAScript 2015 also has a new form of mutable reference declaration, let, which is essentially a drop-in replacement for var; I advise to embrace it at any opportunity.

A more "convenience"-type change is to add string interpolation to the language using backticks (the leftward-leaning single quote that usually appears underneath the tilde on U.S. keyboards) instead of single-quotes or double-quotes:

JavaScript Copy

```javascript
const speaker = { name: "Brian Randell", rating: 5 };
console.log(`Speaker ${speaker.name} got a rating of ${speaker.rating}`);
```

Like C#-style string interpolation, ECMAScript 2015 will interpret the expression inside the braces, attempting to convert it to a string and inserting it into the resulting string.

One of the more important changes to the language has been to embrace "block-scoping"; previously, within JavaScript, variables scoped to functions, not to arbitrary code blocks. This meant that any variable declared anywhere inside of a function was accessible throughout the entire function, not just the block in which it was declared, which was confusing and a subtle source of bugs.

An interesting side effect of this block scoping is that ECMAScript now gains locally declared functions, similar to that proposed for C# 7:

JavaScript Copy

```javascript
{
  function foo () { return 1 }
  foo() === 1
  {
    function foo () { return 2 }
    foo() === 2
  }
  foo() === 1
}
```

Here, I define a function foo in the block, to return the value 1. Then, for no particular reason, I introduce a new scope block, define a new definition of foo, and demonstrate that when that scope block closes, the definition of foo returns to the previous version —which is exactly what almost every other language on the face of the planet does already.

However, idiomatically, ECMAScript doesn't use locally nested functions; instead, it prefers a more functional-style programming idiom, defining a variable reference to point to an anonymous function definition, and use that. In support of that, ECMAScript 2015 now supports arrow functions, which use a syntax almost identical to that of C#:

JavaScript Copy

```javascript
const nums = [1, 2, 3, 4, 5];
nums.forEach(v => {
  if (v % 2 == 0)
    console.log(v);
});
```

(Remember that you added the function forEach to arrays as part of the previous ECMAScript standard.) This will, as assumed, print out the even numbers in the array. If, on the other hand, I want to construct even numbers out of the source array, I can use the built-in "map" and an arrow function to do essentially the same thing:

JavaScript Copy

```javascript
const nums = [1, 2, 3, 4, 5];
const evens = nums.map(v => v * 2);
```

Arrow functions are a long-awaited change and it's reasonable to expect that most JavaScript-based code will adopt them aggressively.

Promises

Of the many solutions that were floated through the ECMAScript community to help address some of the complexity around the Node.js callback-style of programming, one of the recurring themes was that of "promises"—essentially, a library-based approach that transforms the callbacks into something that looked more serial in nature. Several different promises-based libraries were popular within the community, and the ECMAScript committee eventually chose to standardize on one, which it now simply refers to as "Promises." (Note the uppercase in the name; this is also the name of the principal object used to implement them.) Using ECMAScript 2015 Promises can look a little weird at first, compared to standard synchronous programming, but for the most part, it makes sense.

Consider, for a moment, application code that wants to invoke a library routine that uses a Promise to be able to do some things in the background:

JavaScript Copy

```
msgAfterTimeout("Foo", 100).then(() =>
  msgAfterTimeout("Bar", 200);
).then(() => {
  console.log(`done after 300ms`);
});
```

Here, the idea is that msgAfterTimeout is going to print "Hello, Foo" after a 100 ms timeout, and afterward, do the same thing again ("Hello, Bar" after 200 ms), and after that, simply print a message to the console. Notice how the steps are connected using then function calls—the returned object from msgAfterTimeout is a Promise object and then defines the function invoked when the initial Promise has completed execution. This explains the name—the Promise object is, in effect, promising to invoke the then function when the initial code is complete. (In addition, what happens if an exception is thrown from within the function? The Promise allows you to specify a function executed in that case, using the catch method).

In the case where you want to run several functions simultaneously and then execute a function after all have finished, you can use Promise.all:

JavaScript Copy

```
const fetchPromised = (url, timeout) => { /* ... */ }
Promise.all([
  fetchPromised("https://backend/foo.txt", 500),
  fetchPromised("https://backend/bar.txt", 500),
  fetchPromised("https://backend/baz.txt", 500)
]).then((data) => {
  let [ foo, bar, baz ] = data;
  console.log(`success: foo=${foo} bar=${bar} baz=${baz}`);
}, (err) => {
  console.log(`error: ${err}`);
});
```

As you can see, the results of each asynchronously executed function will be collected and assembled into an array, which is then passed as the first parameter to the (first) function given to then. (The let statement is an example of "destructuring assignment" in ECMAScript 2015, another new feature; essentially, each element of the data array is assigned to each of those three declared variables and the remainder, if any, thrown away.) Notice, as well, that then is passed a second function, which is used in the event that there's an error in executing any of the async functions.

It's definitely a different style of programming, but not unusual for anyone who's spent time working with the various C# asynchronous mechanisms, a la PLINQ or TPL.

Library

ECMAScript 2015 has added a few important things to the standard library (which all ECMAScript environments are supposed to provide, be they browser or server) beyond the Promise. In particular, they've added a Map (key/value store, similar to the .NET Dictionary<K,V> type) and a Set (a no-duplicates "bag" of values), both of which are available without requiring any sort of import:

JavaScript Copy

```javascript
const m = new Map();
m.set("Brian", 5);
m.set("Rachel", 5);
console.log(`Brian scored a ${m.get("Brian")} on his talk`);
const s = new Set();
s.add("one");
s.add("one"); // duplicate
s.add("one"); // duplicate
console.log(`s holds ${s.size} elements`);
  // Prints "1"
```

In addition, several more standard functions are being added to various object prototypes already in the ECMAScript environment, such as find on Arrays, and some numeric valuation methods (such as isNAN or isFinite) to the Number prototype. For the most part, both these and the earlier Map and Set types were already present in the community as third-party packages, but bringing them in to the standard ECMAScript library will help cut down on some of the package-dependencies that litter the ECMAScript landscape.

Modules

Probably one of the most significant changes, at least from the perspective of building a MEAN-based application, is the adoption of a formal module system. Previously, as I discussed almost a year ago, a MEAN-based application used the Node.js require function to "import" (interpret/execute) another JavaScript file and return an object for use. This meant that when a MEAN developer wrote the following line, the express object returned by evaluating a JavaScript file stored in a subdirectory inside of node_modules, which was installed via the use of npm:

JavaScript Copy

```
var express = require('express');
```

To make this work, several conventions had to be in place, and it worked, but the lack of formality distinctly hindered the language and ecosystem's forward progress. In ECMAScript 2015 new keywords were introduced to formalize much of this.

It's a bit circular to explain, so let's start with a simple example: two files, one called app.js, and the library it wants to use, called math. The math library will be a non-npm library (for simplicity), and will have two values it exports: the value of pi (3.14), called PI, and a function to sum up an array of numbers, called sum:

JavaScript Copy

```
//  lib/math.js
export function sum (x, y) { return x + y };
export var pi = 3.141593;
//  app.js
import * as math from "lib/math";
console.log("2PI = " + math.sum(math.pi, math.pi));
```

Notice that in the library, the symbols made accessible to clients are declared with the "export" keyword, and when referencing the library, you use the keyword "import." More commonly, however, you want to import the symbols from the library as top-level elements themselves (rather than as members), so the more common usage will likely be this:

JavaScript Copy

```
//  app.js
import {sum, pi} from "lib/math"
console.log("2π = " + sum(pi, pi));
```

This way, the imported symbols can be used directly, rather than in their member--scoped form.

Wrapping Up

There's a great deal more hiding in the ECMAScript 2015 standard; readers who haven't seen any of these features before now should definitely check out any of the (ever-growing list of) resources describing the new standard found on the Internet. The official ECMA standard is available at bit.ly/1xxQKpl (it lists the current specification as 7th edition, called ECMAScript 2016, largely because ECMA has decided to go to a yearly cadence on new changes to the language). However, for a less "formal" description of the language, readers are encouraged to check out es6-features.org, which provides a list of the new language features and their comparison to what was present before within the language.

In addition, while Node.js is almost entirely compliant with the feature set of ECMAScript 2015, other environments aren't, or are in various states of support. For those environments, that aren't quite up to a complete level of support, there are two "transpiler" tools—the Google-sponsored Traceur compiler and the Babel compiler. (Both are, of course, just an npm away.) Of course, Microsoft's own TypeScript is amazingly close to what ECMAScript 2015 ended up being, meaning that one could adopt TypeScript today and be almost line-for-line compliant with ECMAScript 2015 if/when converting to ECMAScript 2015 is needed or desirable.

All these features will become more obvious as you start working with MEAN tools that make use of them, so don't stress for now if they don't make sense yet. In the meantime … happy coding!

Ted Neward *is a Seattle-based polytechnology consultant, speaker and mentor. He has written more than 100 articles, is an F# MVP and has authored and coauthored a dozen books. Reach him at ted@tedneward.com if you're interested in having him come work with your team, or read his blog at blogs.tedneward.com .*

Thanks to the following technical expert for reviewing this article: Shawn Wildermuth

Discuss this article in the MSDN Magazine forum