

01/31/2019 • 9 minutes to read

In this article

[Edge.js](#)

[Hello, Edge](#)

[Hello, SQL Server](#)

[Why, Again?](#)

[Wrapping Up](#)

May 2016

Volume 31 Number 5

[The Working Programmer] How To Be MEAN: Getting the Edge(.js)

By [Ted Neward](#) | May 2016



Welcome back, “MEANers.” In the previous installment, I added a bit of structure to the otherwise structureless environment that is JavaScript, Node and MongoDB, by adding the MongooseJS library to the software stack I’ve slowly been building. This put some “schema” around the different collections that the Node/Express middleware was receiving and storing, which is nice because it helps avoid some common human-inspired errors (such as searching for “fristName” instead of the actual field “firstName”). Best of all, MongooseJS is entirely code-side, which means that for all practical purposes, you now have the best of both worlds, at least as far as the database is concerned—“schemaless” in the database (making it far easier to refactor) and “schemaful” in the code (making it far less likely that a typo will mess things up).

But, if I can take a personal moment here, I must admit that I miss the Microsoft .NET Framework. Or, to be more specific, I miss some of the very cool things that the .NET ecosystem has available within it. Particularly, when I’m executing on the Microsoft


Azure cloud, where a number of organizations are going to have some small (or very large) investment in the .NET “stack,” it seems a little out of place to be talking so much about JavaScript, if all of that .NET stuff remains out of reach. Or, at least, out of reach except for doing some kind of long-haul HTTP-style request, which seems kind of silly when you’re operating inside the same datacenter.

Fortunately, we have an edge. Or, to be more specific, Edge.js.

Edge.js

The Edge.js project is seriously one-of-a-kind in a lot of ways, most notably that it seeks to very directly address the “platform gap” between .NET and Node.js. Hosted at bit.ly/1W7xJmo, Edge.js deliberately seeks to make each platform available to the other in a very code-friendly way to each.

For example, getting a Node.js code sample to call a .NET function looks like this:

JavaScript	 Copy
<pre>var edge = require('edge'); var helloWorld = edge.func(function () {/* async (input) => { return ".NET Welcomes " + input.ToString(); } */}); helloWorld('JavaScript', function (error, result) { if (error) throw error; console.log(result); });</pre>	

As you can see, programmatically, this isn’t difficult: Pass a function literal to the `edge.func` method and inside that function literal include the .NET code to invoke as the body of a comment.

Yes, dear reader, a comment. This isn’t so strange when you realize:

- It can’t be literal C# syntax or the Node interpreter wouldn’t recognize it as legitimate program syntax (because, after all, the Node interpreter is a JavaScript interpreter, not a C# interpreter).
- Unlike a compiled program, the interpreter has access to the full body of whatever source code is defined there, rather than just what the compiler chose to emit.

Note that this isn’t limited to just C#, by the way; the Edge.js project lists several other languages that can be used as the “target” of an Edge call, including F#, Windows PowerShell, Python or even Lisp, using .NET implementations of each of those languages. My favorite, of course, is F#:

JavaScript

 Copy

```
var edge = require('edge');
var helloFs = edge.func('fs', function () {/*
    fun input -> async {
        return "F# welcomes " + input.ToString()
    }
*/});
helloFs('Node.js', function (error, result) {
    if (error) throw error;
    console.log(result);
});
```

Note the key difference is an argument slipped in front of the function literal, indicating which language is being passed in that function comment.

The key here is to realize that the body of the function—whether written in C# or in F#—is of a specific .NET-type signature: `Func<object, Task<object>>`. The asynchrony here is necessary because remember that Node prefers callbacks to direct sequential execution in order to avoid blocking the main Node.js event loop.

Edge.js also makes it relatively easy to invoke these functions on compiled .NET DLLs. So, for example, if you have a compiled assembly of .NET code that wants to be invoked, Edge.js can invoke it so long as the assembly name, type name and method name are provided as part of the “func” call:

JavaScript

 Copy

```
var helloDll = edge.func({
    assemblyFile: "Echo.dll",
    typeName: "Example.Greetings",
    methodName: "Greet"
});
```

If the type signature of `Greet` is a `Func<object, Task<object>>`, such as the one shown in **Figure 1**, then Node.js can call it using the same invocation pattern (passing in input arguments and a function callback), as shown for the other examples.

Figure 1 An Edge.js-Compatible .NET Endpoint

C#

 Copy

```
using System;
using System.Threading.Tasks;
namespace Example
{
    public class Greetings
    {
        public async Task<object> Greet(object input)
```

```
{
    string message = (string)input;
    return String.Format("On {0}, you said {1}",
        System.DateTime.Now,
        Message);
}
}
```

It's also possible to go the other way—to arrange .NET code to call into Node.js packages—but because the goal here is to work Node.js on the server side, I'll leave that as an exercise to the interested reader. (By the way, all the Edge.js stuff is much easier to work with from a Windows machine than a Mac; trying to get it to work on my Mac during the writing of this column was definitely too time-consuming, all things considered, so this is one case where the Windows experience definitely trumps the Mac for Node.js-related development.)

I want to give this a quick hello-world-style spin before doing anything more complicated.

Hello, Edge

First things first, again, all of this needs to work in Microsoft Azure (because that's my chosen target deployment environment), so make sure to “npm install --save edge” so that it'll be tracked in the package.json manifest when it gets committed to Azure. Next, I add the “helloWorld” function to the app.js code, and set up a quick endpoint so that I can “GET” on it and get that greeting back via HTTP, as shown in **Figure 2**. And, sure enough, sending a GET to `msdn-mean.azurewebsites.net/edgehello` brings back:

JavaScript	 Copy
<pre>{"message": ".NET Welcomes Node, JavaScript, and Express"}</pre>	

Figure 2 Adding the “helloWorld” Function

JavaScript	 Copy
<pre>var helloWorld = edge.func(function () {/* async (input) => { return ".NET Welcomes " + input.ToString(); } */}); var edgehello = function(req, res) { helloWorld('Node, JavaScript, and Express', function (err, result) { if (err) res.status(500).jsonp(err); else res.status(200).jsonp({ message: result }); }); };</pre>	

```
};  
// ...  
app.get('/edgehello', edgehello);
```

Hello, SQL Server

One question that periodically arises whenever I talk about the MEAN stack with .NET developers is about MongoDB. A fair number of people don't like the idea of giving up their SQL Server, particularly when running on Azure. Of course, the Node.js community has built several relational database access APIs and SQL Server is just a TDS connection away from any of those, but Edge.js actually has a pretty fascinating solution to that particular problem (once you "npm install --save edge-sql," to pull in the Edge-SQL package):

JavaScript

 Copy

```
var edge = require('edge');  
var getTop10Products = edge.func('sql', function () { /*  
    select top 10 * from Products  
*/});  
getTop10Products(null, function (error, result) {  
    if (error) throw error;  
    console.log(result);  
    console.log(result[0].ProductName);  
    console.log(result[1].ReorderLevel);  
});
```

This code is assuming that the Azure environment has an environment variable called `EDGE_SQL_CONNECTION_STRING` set to the appropriate SQL Server connection string (which, in this case, would presumably point to the SQL Server instance running in Azure).

That's arguably simpler than working with just about anything else I've seen, to be honest. It probably won't replace Entity Framework any time soon, granted, but for quick access to a SQL Server instance, perhaps as part of a "polypraeclusio" ("multiple storage") approach that uses SQL Server for storing the strictly schemaed relational data and MongoDB for the schemaless JSON-ish data, it's actually pretty elegant.

Why, Again?

Given that most developers generally look askance at any solution that requires them to be conversant in multiple languages at the same time, it's probably worth digging a little deeper into when and how this might be used.

The obvious answer is that for most greenfield kinds of projects, with no legacy code support required, the general rule would be to stay entirely inside of one language/platform or the other: either stick with .NET and use Web API and so on, or go “whole hog” into Node.js, and rely on the various libraries and packages found there to accomplish the goals at hand. After all, for just about anything you can think of doing in the .NET world, there’s likely an equivalent package in the npm package repository. However, this approach comes with a few caveats.

First, the .NET ecosystem has the advantage of having been around a lot longer and, thus, some of the packages there being more battle-tested and trustworthy. Many of the npm packages are still flirting with highly dubious version numbers; managers have a hard time trusting anything with a version number starting with 0, for example.

Second, certain problems lend themselves better to certain programming approaches or environments; case in point, the F# language frequently “makes more sense” to those with a mathematical background and makes it easier sometimes for them to write certain kinds of code. Such was the case a few years ago when I wrote the Feliza library (as part of the series demonstrating how to use SMS in the cloud using the Tropo cloud); while it could have been written in C#, F#’s pattern-matching and “active patterns” made it much easier to write than had I done it in C#. The same is true for JavaScript (perhaps more so).

Last, and perhaps most important, is that sometimes the environment in which the code is executing just naturally favors one platform over another. For example, numerous organizations that host applications in Azure use Active Directory as their authentication and authorization base; as a result, they’ll want to continue to use Active Directory for any new applications written, whether in .NET or Node.js. Accessing Active Directory is generally much simpler and easier to do from a .NET environment than anything else, so the Edge.js library offers a convenient “trap door,” so to speak, to make it much easier to access Active Directory.

Wrapping Up

It’s been a little lighter this time around, largely because the Edge.js library takes so much of the work out of working interoperably with the .NET environment. And it opens a whole host of new options for the Azure MEANer, because now you have access to not just one but two rich, full ecosystems of tools, libraries and packages.

Note that there’s a whole column waiting to be written going the other direction, too, by the way: In many cases, certain kinds of applications are much easier to write using the various packages available in the npm repository, that Edge.js now opens up to the traditional .NET developer. And with the recent open source release of Chakra—the

Microsoft JavaScript core that can “plug in” to the Node environment as a drop-in plug-in—this opens up even more opportunities to use JavaScript as part of a “standard” .NET application, including even the opportunity to host a JavaScript interpreter at the center of your application. This has its own interesting implications when you stop to think about it.

There are a few more things I want to discuss before moving off the server completely, but I’m out of space, so for now ... happy coding!

Ted Neward *is a Seattle-based polytechnology consultant, speaker and mentor. He has written more than 100 articles, is an F# MVP, INETA speaker, 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](#)