James M Snell Follow
IBM Technical Lead for Node.js
2 days ago · 15 min read

Node.js, TC-39, and Modules

This week I attended my first TC-39 meeting. For those unaware, TC-39 is the designator of the ECMA working group that defines the ECMAScript Language (or "JavaScript" as it is more commonly known). It is the forum where all of the various nuances and details of the JavaScript language are hammered out (often painfully) and worked through in order to ensure that the JavaScript programming language continues to evolve and continues to meet the needs of developers.

The reason I attended the TC-39 meeting this week is fairly simple: one of the newer JavaScript language features defined by TC-39—namely, Modules—has been causing the Node.js core team a bit of trouble. We (and by we I mean mostly Bradley Farias—@bradleymeck on Twitter) have been trying to figure out how to best implement support for ECMAScript Modules (ESM) in Node.js without causing more trouble and confusion than it would be worth.

The issue has not really been that we cannot implement ESM in Node.js the way the specification is currently defined, it's that doing so to the letter of the specification would mean a reduction in expected functionality and a suboptimal experience for Node.js developers. We very much want to make sure that an implementation of ESM in Node.js is both optimized and usable. Because of the complexity of the issues involved, sitting down face to face with the members of TC-39 was deemed to be the most productive path forward. Fortunately, I think we made some significant progress.

To understand where things are at, and where things are going, however, let me take some time to explain what fundamental issues have been causing us the most concern.

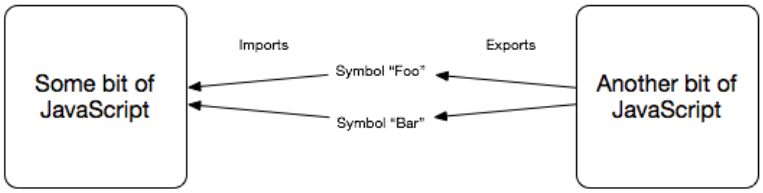
First, however, a warning: a lot of the following will be an oversimplification of what is actually happening under the covers in the code. This is intended primarily to provide an overview and not an in depth treatise on module systems.

Then, another warning: everything here is based on my own perception of the conversation with TC-39. It's entirely possible that I've got some details wrong and it's entirely possible and likely that the conversation will continue to evolve and that things will ultimately look much different than what I describe here. I am writing this up only to provide a snapshot of what is being discussed.

ECMAScript Modules vs. CommonJS: Or... What is a Module?

It turns out, Node.js and TC-39 have very different ideas of what a “module” is, how those are defined, and how they are loaded into memory and used.

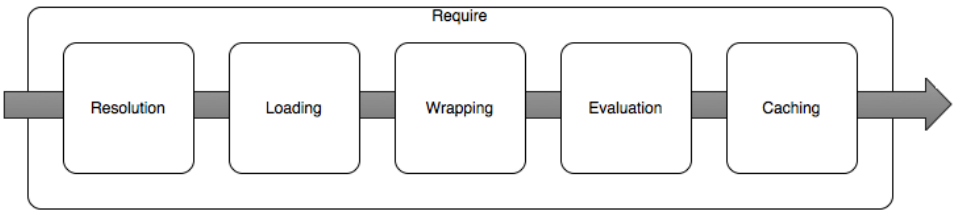
From nearly the beginning, Node.js has had a module system that is derived from a fairly loosely defined specification called “CommonJS”.




```
const mod = require('dep');
mod.Foo();
console.log(mod.Bar);
```

```
module.exports.Foo = function () { … };
module.exports.Bar = 1;
```

The short version is that symbols exported by one JavaScript file (things like functions and variables) are made available for use by another JavaScript file. In Node.js, this is accomplished using the *require()* function. When a call like *require(“foo”)* is called within Node.js, there is a very specific sequence of steps performed.



Step one is to resolve the specifier *“foo”* into an absolute file path to some kind of artifact that Node.js understands. This resolution process involves multiple internal steps that essentially walk the local file system for any native module, JavaScript file, or JSON document that happens to match. The result of the resolution step is an absolute file path from which the artifact specified by *“foo”* can be loaded into Node.js and used.

Loading is determined entirely by what kind of thing the absolute file path produced by the resolution step is pointing to. For instance, if the thing pointed at by the resolved path is a Node.js native module, then loading involves dynamically linking the referenced shared library into the current Node.js process. If the thing pointed at is a JSON file or a JavaScript file, the contents of the file are read into memory after the file is verified to exist. It is important to note that Loading the JavaScript is not the same as Evaluating the JavaScript. The former deals strictly with pulling the string

contents of the file into memory while the latter deals with passing that string off to the JavaScript VM for parsing and evaluation.

If the loaded artifact is a JavaScript file, then Node.js currently assumes that the file is a CommonJS module. What Node.js does next is critical and is often misunderstood by developers creating Node.js applications. Before passing the loaded JavaScript string off to the JavaScript VM for evaluation, the string is *wrapped inside a function*.

For instance, a file "*foo.js*" such as:

```
const m = 1;
module.exports.m = m;
```

Is actually evaluated by Node.js as a function like:

```
function (exports, require, module, __filename, __dirname) {
  const m = 1;
  module.exports.m = m;
}
```

Node.js then uses the JavaScript runtime to evaluate this function. The various "global" artifacts like "*exports*", "*module*", "*__filename*", and "*__dirname*" that are commonly used in Node.js modules are not actual globals in the traditional JavaScript sense. They are, instead, function parameters, whose values are provided to the wrapped function by Node.js when the function is called.

This wrapper function is essentially a *factory* method. The exports object is a regular JavaScript Object. The wrapper function attaches functions and properties to that exports object. Once the wrapper function returns, the exports object is cached then returned as the return value for the *require()* method.

The key concept to understand for this particular discussion is that there is no way of determining *in advance* what symbols are going to be exported by the CommonJS module until after the wrapper function is evaluated.

This is a critical difference between CommonJS modules and ECMAScript modules because while exports of a CommonJS module are defined dynamically while the wrapper function is being evaluated, the exports of an ESM are defined *lexically*. That is, the symbols exported by an ESM are

determined when the JavaScript code is being *parsed* before it is actually evaluated.

For example, given the following simple ECMAScript module:

```
export const m = 1;
```

When this code is *parsed*, but before it is evaluated, an internal structure called a Module Record is created. Within this Module Record, among other key bits of information, is a *static* listing of the symbols that are exported by the module. These are identified by the parser looking for the use of the *export* keyword. For lack of a better term, the symbols in the Module Record are essentially *pointers* to things that do not yet exist. Only after this Module Record is constructed is the module code actually evaluated. While there are many details lurking in here that I am glossing over, the key point is that determining what symbols are exported by an ESM occurs *before* evaluation.

When code uses an ECMAScript module, it uses an *import* statement:

```
import {m} from “foo”;
```

This code basically says, “I’m going to use the m symbol exported by module ‘foo’”.

This statement is a *lexical* statement that is used to establish a link between the importing script and the “*foo*” module when the code is *parsed*. The way the ECMAScript module specification is written today, this link must be validated before any of the code is evaluated—meaning that the implementation must make sure that the symbol “m” really is exported by “foo” before either of the two JavaScript files can be evaluated at all.

For anyone familiar with strongly typed object oriented programming languages like Java or C++, this should be readily familiar because it is analogous to working with an object through an interface. The symbols exported are verified and linked before execution, and errors will be thrown if the symbols are not actually fulfilled during the execution step.

For Node.js, a challenge arises when “*foo*” is not a ESM with a lexically defined set of exports, but a CommonJS module with a dynamically defined set of exports. Specifically: when I say *import {m} from “foo”*, ESM currently requires that it is possible to determine that *m* is exported by “*foo*” before

evaluation; but as we've already seen, since "*foo*" is a CommonJS module, it is not possible to determine that *m* is exported until *after* evaluation. The end result is that named exports and imports from CommonJS, a critically important feature of ECMAScript modules, simply would not be possible under the currently defined ESM specification.

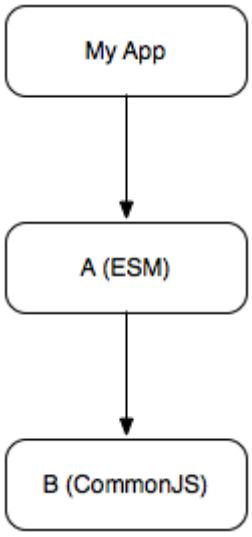
This is not particularly ideal so we (the Node.js people) went back to TC-39 to ask if some changes could be made in the spec. At first, we were a bit afraid to ask. It turns out, however, that TC-39 cares an awful lot about making sure that Node.js can implement ESM's effectively and a number of changes to the specification are being looked at in order to make things work better within the Node.js environment.

Order of operations

One specific change being proposed is to account for dynamically defined modules. Essentially, when I do *import {m} from "foo"*, and it turns out that "*foo*" is not an ESM with *lexically* defined exports, rather than throwing an error and giving up (which is what the spec currently does) the process would be to put "*foo*" and the importing script into a kind of intermediate pending state, deferring validation of the imported symbols until the dynamic module's code can be evaluated. Once evaluated, the Module Record for the CommonJS module can be completed and the imported links validated. This modification to the ECMAScript module standard allows named exports and imports from CommonJS modules to Just Work. (Although, there are a few gotchas with regards to a few circular dependency edge cases).

Let's walk through a couple of examples.

I have an application that depends on ESM A, which depends on CommonJS module B.



The code for my application (*myapp.js*) is:

```
const foo = require('A').default
foo()
```

The code of A is:

```
import {log} from "B"
export default function() {
  log('hello world')
}
```

The code of B is:

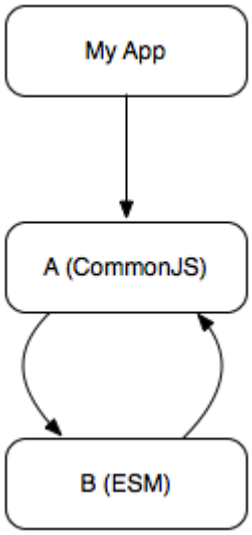
```
module.exports.log = function(msg) {
  console.log(msg);
}
```

When I run *node myapp.js*, the call to require('A') would detect that the thing that is being loaded is an ESM (see below for how this detection would likely be done). Rather than loading the module using the wrapper function that is currently used for CommonJS modules, Node.js would use the ECMAScript module specification to parse, initialize and evaluate "A". When the code for "A" is parsed, and the Module Record is created, it would detect that "B" is not an ESM, so the validation step that verifies that log is exported by "B" would be pending. The ESM loader would then begin it's evaluation phase. This would first evaluate B using the existing CommonJS wrapper function the results of which would be passed back to the ESM loader to complete construction of the Module Record. Second, it would evaluate the code for "A", using that completed Module Record.

What about switching the order of the dependencies. Let's say that A is a CommonJS and B is an ESM. Here, things just work without any special things being done, because as illustrated in the example above, it will be possible to require() an ESM.

For the overwhelming majority of basic use cases, this loading model should work just fine. Where it starts to get tricky is when a dependency cycle exists between modules. Anyone who has used CommonJS modules with circular dependencies before knows that there are some rather weird edge cases that

can creep up depending on the order in which those modules are loaded. A number of the same kinds of issues would exist in the case where a dependency cycle exists between a CommonJS module and ESM.



The code for myapp.js remains the same as above. However, A depends on B which in turn depends on A.

The code for A is:

```
const b = require('B')
exports.b = b.foo()
exports.a = 1
```

The code for B is:

```
import {a} from "A"
export const foo () => a
```

This is a fairly contrived case primarily to illustrate the issue. This type of cycle becomes largely impossible to fulfill because when the ESM “B” is linked and evaluated, the symbol “a” has not yet been defined and exported by CommonJS module “A”. This type of case would likely *have* to be treated as a reference error.

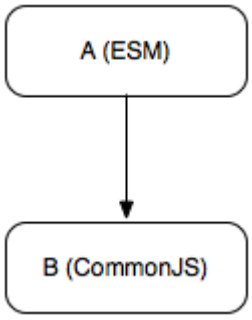
However, if we changed the code for B to:

```
import A from “A”
export foo () => A.a
```

The circular dependency works because when a CommonJS module is imported using the import statement, the module.exports object becomes the *default* export. The ESM code in this case is linking to the *default* export of the CommonJS module rather than to the *a* symbol.

To put it more succinctly: named imports from a CommonJS module would work only if there is not a dependency cycle between the ESM and the CommonJS module.

Another limitation caused by the differences between CommonJS and ESM is that any mutation to the CommonJS exports after the initial evaluation would not be available as a named import. For instance, suppose ESM A depends on CommonJS module B.



Suppose the code for B is:

```
module.exports.foo = function(name, key) {
  module.exports[name] = key
}
```

When “B” is imported by “A”, the only exported symbols that will be available for use as named imports will be the default symbol and “*foo*”. None of the symbols added to module.exports when calling the foo function would be available as named imports. They would, however, be available via the default export. The following, for instance, should work just fine:

```
import {foo} from “B”
import B from "B"
```

```
foo("abc", 123)
if (B.abc === 123) { /** ... **/ }
```

require() vs import

There's one very clear distinction that needs to be made with regards to *require()* and *import*: while it will be possible to load an ESM using *require()* and it will be possible to import a CommonJS module using *import*, it will not be possible to use the *import* statement from within a CommonJS module; and by default, *require()* will not be available within an ESM.

In other words, if I have a CommonJS Module A, the following code will ***not*** be possible because the import statement will not be usable from within CommonJS:

```
const b = require('B')
import c from "C"
```

If you are operating within a CommonJS module, the right way to load and use an ESM will be to use require:

```
const b = require('B')
const c = require('C')
```

From within an ESM, require() will only be available and usable if it is specifically imported. The exact specifier used to import require() has yet to be determined but essentially it would be something like:

```
import {require} from "nodejs"
require("foo")
```

However, because it will be possible to import directly from a CommonJS module, there should be exceedingly few reasons to do this.

As a side node: the Node.js folks had a number of other concerns such as whether or not loading of ESM's would always *have* to be asynchronous, which would require the use of Promises throughout the entire dependency

graph. TC-39 assured us (and the changes described above allow) that loading would not have to be asynchronous. This is a very good thing.

What about import()

There is a proposal being put before TC-39 that would introduce a new *import()* function. This would be a distinctly different thing than the import statement shown in the examples above. Consider the following example:

```
import {foo} from “bar”
import(“baz”).then((module)=>{/*…*/}).catch((err)=>{/**…*/})
```

The first import statement is lexical. As described earlier, it is processed and validated when the code is *parsed*. The import() function, on the other hand, 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.

Because the *import()* function returns a Promise, things like *await import(“foo”)* will be possible. However, it is important to note that *import()* is far from complete within TC-39 and has not yet matured. It’s also not entirely clear yet if Node.js will be able to implement fully asynchronous loading using the *import()* function.

Detection of CommonJS vs. ESM

Whether or not code uses *require()*, *import* or *import()* to load modules, it is necessary to be able to detect the kind of thing that is being imported so that Node.js can know the appropriate way of loading and processing it.

Traditionally, The Node.js implementation of the *require()* function has relied on file extensions to differentiate how to load different types of things. For instance, **.node* files are loaded as native modules, **.json* files are simply passed through JSON.parse, and **.js* files are handled as CommonJS modules.

With the introduction of ESM, a mechanism is required to differentiate between CommonJS modules and ESM. There have been a couple suggested approaches.

One proposal is to ensure that a JavaScript file can be unambiguously parsed as either an ESM or something else. In other words, when I parse a bit of

JavaScript, the fact that it is an ESM or not should be obvious by the result of the parse operation. This approach is called “unambiguous grammar”. Unfortunately, it’s a bit trickier to accomplish that it may appear.

Another proposal that has been considered is adding metadata to the *package.json* file. If some specific value is in the package.json file, then the module would be loaded as an ESM rather than as a CommonJS module.

A third proposal is to use a new file extension (**.mjs*) to identify ECMAScript modules. This is the approach that most closely matches what Node.js already does today.

For instance, suppose I have an application script *myapp.js* and an ESM module defined in a separate file.

Using the unambiguous grammar approach, Node.js should be able to parse the JavaScript in the second file and determine automatically that it’s dealing with an ESM. With this approach, the ESM file could use the **.js* file extension and things would just work. As I said, however, unambiguous grammar is quite a bit trickier to get right and there are a number of edge cases that make it difficult to achieve.

Using the *package.json* approach, the ESM would either have to be bundled in it’s own directory (essentially it’s own package) or there would have to be a *package.json* in the root that contains some bit of metadata indicating that the JavaScript containing the ESM file is, in fact, an ESM. This approach is less than ideal because of the additional processing of the *package.json* that is required.

Using the **.mjs* file extension approach, the ESM code is put into a file like *foo.mjs*. After Node.js resolves the specifier into the absolute filename, it would look at the file extension just as it currently already does with native addons and JSON files. If it sees the **.mjs* file extension, it knows to load and process the thing as an ESM. If it sees **.js*, however, it would fallback and load the thing as a CommonJS module.

At the current point in time, the **.mjs* file extension is looking like the most viable option unless all of the various edge cases for unambiguous grammar can be worked out.

Idempotency Concerns

Generally speaking, calling *require(‘foo’)* multiple times will return the exact same instance of the module. The object that is returned, however, is mutable, and it is possible for modules to modify other modules either by monkeypatching individual methods and symbols, or by replacing the

functionality entirely. This type of thing is extremely common in the Node.js ecosystem currently.

For example, suppose myapp.js has two dependencies A and B. Both of which are CommonJS modules. A also depends on B in order to extend it.

The code for myapp.js is:

```
const A = require('A')
const B = require('B')
B.foo()
```

The code for A is:

```
const B = require('B')
const foo = B.foo


B.foo = function() {
  console.log('intercepted!')
  foo()
}
```

The code for B is:

```
module.exports.foo = function() {
  console.log('foo bar baz')
}
```

In this scenario, require('B') when called within A returns a different result than require('B') called within myapp.js.

With ECMAScript Modules, this type of monkeypatching across modules *is not as easy*. The reason is twofold: A) imports are linked before evaluation and B) imports are required to be idempotent—always returning the exact same immutable set of symbols each time the import is called within a given context. What this means in a practical sense is that ESM A cannot easily monkeypatch ESM B when named imports are used.

The effect of this rule is equivalent to the following code in myapp.js

```
const B = require('B')
const foo = B.foo
const A = require('A')
foo()
```

Here, module A still modifies foo in B, but because the reference to foo was grabbed before that modification, the call to foo() invokes the original function, rather than the modified one. Within an ESM, there would be no way of importing B that would return the modifications made by A.

There are a wide variety of scenarios where this idempotency rule causes issues. Mocking, APM, and spying for testing purposes are primary examples. Fortunately, there are a number of ways this limitation can be addressed. One approach is to add hooks into the loading phase that would allow an ESM's exports to be wrapped. Another is for TC-39 to allow loaded ESM's to be replaced after they are loaded. Several mechanisms are being considered here. The good news is that while intercepting ESM's will be different than intercepting CommonJS modules, interception *will* be possible.

Lots more to do

There is a *ton* of additional work to do and everything discussed above is not yet final in any sense. There are many details to work out and things could end up looking very different in the end. The important thing is that Node.js and TC-39 are working together on figuring all of this out, which is an excellent and very welcome step in the right direction.