Using Node.js require vs. ES6 import/export

Asked 6 years, 3 months ago Active 1 month ago Viewed 680k times

1147

331

In a project I'm collaborating on, we have two choices on which module system we can use:

1. Importing modules using `require`, and exporting using `module.exports` and `exports.foo`.
2. Importing modules using ES6 `import`, and exporting using ES6 `export`

Are there any performance benefits to using one over the other? Is there anything else that we should know if we were to use ES6 modules over Node ones?

javascript node.js ecmascript-6 babeljs

Share Improve this question Follow

edited Oct 14 '17 at 12:47
sepehr
14.4k 6 74 113

asked Jul 11 '15 at 7:19
kpimov
12.1k 3 10 18

11 `node --experimental-modules index.mjs` lets you use `import` without Babel and works in Node 8.5.0+. You can (and should) also publish your npm packages as native ESModule, with backwards compatibility for the old `require` way. – Dan Dascalescu Jan 26 '19 at 1:14

No necessary to use .mjs files, just use type: "module" in your package.json and put extension when importing only for your project files, that's it – Máxima Alekz May 24 at 21:13

10 Answers

Active Oldest Votes

865

Update

Since Node v12 (April 2019), support for ES modules is enabled by default, and since Node v15 (October 2020) it's stable (see here). Files including node modules must either end in `.mjs` or the nearest `package.json` file must contain `"type": "module"`. The Node documentation has a ton more information, also about interop between CommonJS and ES modules.

Performance-wise there is always the chance that newer features are not as well optimized as existing features. However, since module files are only evaluated once, the performance aspect can probably be ignored. In the end you have to run benchmarks to get a definite answer anyway.

ES modules can be loaded dynamically via the `import()` function. Unlike `require`, this returns a promise.

Previous answer

> Are there any performance benefits to using one over the other?

Keep in mind that there is no JavaScript engine yet that natively supports ES6 modules. You said yourself that you are using Babel. Babel converts `import` and `export` declaration to CommonJS (`require` / `module.exports`) by default anyway. So even if you use ES6 module syntax, you will be using CommonJS under the hood if you run the code in Node.

Share Improve this answer Follow

edited Sep 7 at 9:30
Nico Schlömer
40.7k 21 149 199

answered Jul 12 '15 at 12:38
Felix Kling
728k 166 1025 1088

16 I tried to use `ES6 import` with `require` but they worked differently. CommonJS exports the class itself while there is only one class. ES6 exports like there are multiple classes so you have to use `.ClassName` to get the exported class. Are there any other differences which actually effects the implementation – Thellimist Dec 19 '15 at 1:04

87 @Entei: Seems like you want a default export, not a named export. `module.exports = ...;` is equivalent to `export default ....` `exports.foo = ...` is equivalent to `export var foo = ...`; – Felix Kling Dec 19 '15 at 1:08

13 It's worth noting that even though Babel ultimately transpiles `import` to CommonJS in Node, used alongside Webpack 2 / Rollup (and any other bundler that allows ES6 tree shaking), it's possible to wind up with a file that is significantly smaller than the equivalent code Node crunches through using `require` exactly *because* of the fact ES6 allows static analysis of import/exports. Whilst this won't make a difference to Node (yet), it certainly can if the code is ultimately going to wind up as a single browser bundle. – Lee Benson Nov 29 '16 at 19:17

5 unless you need to do a dynamic import – chulian Feb 28 '17 at 7:03

3 ES6 Modules are in the latest V8 and are also arriving in other browsers behind flags. See: medium.com/dev-channel/… – Nexii Malthus May 25 '17 at 12:17

206

There are several usage / capabilities you might want to consider:

Require:

- You can have dynamic loading where the loaded module name isn't predefined /static, or where you conditionally load a module only if it's "truly required" (depending on certain code flow).
- Loading is synchronous. That means if you have multiple `require`s, they are loaded and processed one by one.

ES6 Imports:

- You can use named imports to selectively load only the pieces you need. That can save memory.
- Import can be asynchronous (and in current ES6 Module Loader, it in fact is) and can perform a little better.

Also, the Require module system isn't standard based. It's is highly unlikely to become standard now that ES6 modules exist. In the future there will be native support for ES6 Modules in various implementations which will be advantageous in terms of performance.

Share Improve this answer Follow

edited Jul 11 '15 at 14:40

answered Jul 11 '15 at 8:49
Amit
42.6k 8 70 104

20 What makes you think ES6 imports are asynchronous? – Felix Kling Jul 11 '15 at 12:56

5 @FelixKling - combination of various observations. Using JSPM (ES6 Module Loader...) I noticed that when an import modified the global namespace the effect isn't observed inside other imports (because they occur asynchronously.. This can also be seen in transpiled code). Also, since that is the behavior (1 import doesn't affect others) there no reason not to do so, so it could be implementation dependant – Amit Jul 11 '15 at 13:06

39 You mention something very important: module loader. While ES6 provides the import and export syntax, it does not define how modules should be loaded. The important part is that the declarations are statically analyzable, so that dependencies can be determined without executing the code. This would allow a module loader to either load a module synchronously or asynchronously. But ES6 modules by themselves are not synchronous or asynchronous. – Felix Kling Jul 11 '15 at 14:18

I think it's important not to conflate the module system/syntax with the module loader. E.g if you develop for node, then you are likely compiling ES6 modules to `require` anyway, so you are using Node's module system and loader anyway. – Felix Kling Jul 11 '15 at 14:27

70

As of right now ES6 import, export is always compiled to CommonJS, so there is **no benefit** using one or other. Although usage of ES6 is recommended since it should be advantageous when native support from browsers released. The reason being, you can import partials from one file while with CommonJS you have to require all of the file.

ES6 → `import, export default, export`

CommonJS → `require, module.exports, exports.foo`

Below is common usage of those.

ES6 export default

```
// hello.js
function hello() {
  return 'hello'
}
export default hello

// app.js
import hello from './hello'
hello() // returns hello
```

ES6 export multiple and import multiple

```
// hello.js
function hello1() {
  return 'hello1'
}
function hello2() {
  return 'hello2'
}
export { hello1, hello2 }

// app.js
import { hello1, hello2 } from './hello'
hello1()  // returns hello1
hello2()  // returns hello2
```

CommonJS module.exports

```
// hello.js
function hello() {
  return 'hello'
}
module.exports = hello

// app.js
const hello = require('./hello')
hello()   // returns hello
```

```
}
function hello2() {
  return 'hello2'
}
module.exports = {
  hello1,
  hello2
}

// app.js
const hello = require('./hello')
hello.hello1()   // returns hello1
hello.hello2()   // returns hello2
```

Share Improve this answer Follow

edited Feb 21 '20 at 3:47

answered Feb 21 '20 at 3:41
Hasan Sefa Ozalp
3,325 1 23 30

1 You actually can use `Object Destructuring` when using CommonJS require as well. So you could have: `const { hello1, hello2 } = require("./hello");` and it will be **somewhat** similar to using import/export. – Petar Sep 15 '20 at 9:08

4 This is by far the best answer as it provides not only the description, but also the actual code snippets. – IvanD Apr 14 at 17:55

47

The main advantages are syntactic:

- More declarative/compact syntax
- ES6 modules will basically make UMD (Universal Module Definition) obsolete - essentially removes the schism between CommonJS and AMD (server vs browser).

You are unlikely to see any performance benefits with ES6 modules. You will still need an extra library to bundle the modules, even when there is full support for ES6 features in the browser.

Share Improve this answer Follow

edited Jul 4 '16 at 9:21

answered Jul 11 '15 at 8:15
snozza
2,013 12 17

4 Could you clarify why one needs a bundler even when browsers has full ES6 module support? – E. Sundin Jul 3 '16 at 15:15

1 Apologies, edited to make more sense. I meant that the import/export modules feature is not implemented in any browsers natively. A transpiler is still required. – snozza Jul 4 '16 at 9:23

17 It seems a bit contradictory frased to me. If there is **full support** then what is the purpose of the bundler? Is there something missing in the ES6 spec? What would the bundler actually do that isn't available in a **fully supported environment**? – E. Sundin Jul 4 '16 at 22:14

1 As @snozza said..."the import/export modules feature is not implemented in any browsers naively. A transpiler is still required" – robertmain Oct 13 '17 at 1:35

3 You no longer need any extra libraries. Since v8.5.0 (released more than a year ago), `node --experimemntal-modules index.mjs` lets you use `import` without Babel. You can (and should) also [publish your npm packages as native ESModule, with backwards compatibility] for the old `require` way. Many browsers also support [dynamic imports] natively. – Dan Dascalescu Sep 26 '18 at 3:33

40

> Are there any performance benefits to using one over the other?

Join Stack Overflow to learn, share knowledge, and build your career.

Sign up with email Sign up with Google Sign up with GitHub Sign up with Facebook

In other words, current browser engines including V8 cannot import *new JavaScript file* from the *main JavaScript file* via any JavaScript directive.

(We may be still just a few bugs away or years away until V8 implements that according to the ES6 specification.)

This document is what we need, and this document is what we must obey.

And the ES6 standard said that the module dependencies should be there before we read the module like in the programming language C, where we had (headers) `.h` files.

This is a good and well-tested structure, and I am sure the experts that created the ES6 standard had that in mind.

This is what enables Webpack or other package bundlers to optimize the bundle in some *special* cases, and reduce some dependencies from the bundle that are not needed. But in cases we have perfect dependencies this will never happen.

It will need some time until `import/export` native support goes live, and the `require` keyword will not go anywhere for a long time.

What is `require`?

This is `node.js` way to load modules. (https://github.com/nodejs/node)

Node uses system-level methods to read files. You basically rely on that when using `require`. `require` will end in some system call like `uv_fs_open` (depends on the end system, Linux, Mac, Windows) to load JavaScript file/module.

To check that this is true, try to use Babel.js, and you will see that the `import` keyword will be converted into `require`.

CommonJS
require
module.exports
ES6
import
export

Share Improve this answer Follow

edited Jan 29 '17 at 23:52

answered Nov 5 '16 at 15:48
prosti
30.4k 8 144 131

2 Actually, there's one area where performance *could* be improved -- bundle size. Using `import` in a Webpack 2 / Rollup build process can potentially reduce the resulting file size by 'tree shaking' unused modules/code, that might otherwise wind up in the final bundle. Smaller file size = faster to download = faster to init/execute on the client. – Lee Benson Nov 29 '16 at 19:12

2 the reasoning was no current browser on the planet earth allows the `import` keyword natively. Or this means you cannot import another JavaScript file from a JavaScript file. This is why you cannot compare performance benefits of these two. But of course, tools like Webpack1/2 or Browserify can deal with compression. They are neck to neck: gist.github.com/substack/68f8d502be42d5cd4942 – prosti Nov 29 '16 at 19:46

4 You're overlooking 'tree shaking'. Nowhere in your gist link is tree shaking discussed. Using ES6 modules enables it, because `import` and `export` are static declarations that import a specific code path, whereas `require` can be dynamic and thus bundle in code that's not used. The performance benefit is indirect-- Webpack 2 and/or Rollup can *potentially* result in smaller bundle sizes that are faster to download, and therefore appear snappier to the end user (of a browser). This only works if all code is written in ES6 modules and therefore imports can be statically analysed. – Lee Benson Nov 30 '16 at 10:46

2 I updated the answer @LeeBenson, I think if we consider the native support from browser engines we cannot compare yet. What comes as handy three shaking option using the Webpack, may also be achieved even before we set the CommonJS modules, since for most of the real applications we know what modules should be used. – prosti Dec 5 '16 at 21:57

Join Stack Overflow to learn, share knowledge, and build your career.

Sign up with email Sign up with Google Sign up with GitHub Sign up with Facebook

performance. This is only valid in a browser context, of course. – Lee Benson Dec 6 '16 at 7:42

36

Using ES6 modules can be useful for 'tree shaking'; i.e. enabling Webpack 2, Rollup (or other bundlers) to identify code paths that are not used/imported, and therefore don't make it into the resulting bundle. This can significantly reduce its file size by eliminating code you'll never need, but with CommonJS is bundled by default because Webpack et al have no way of knowing whether it's needed.

This is done using static analysis of the code path.

For example, using:

```
import { somePart } 'of/a/package';
```

... gives the bundler a hint that `package.anotherPart` isn't required (if it's not imported, it can't be used- right?), so it won't bother bundling it.

To enable this for Webpack 2, you need to ensure that your transpiler isn't spitting out CommonJS modules. If you're using the `es2015` plug-in with babel, you can disable it in your `.babelrc` like so:

```
{
  "presets": [
    ["es2015", { modules: false }],
  ]
}
```

Rollup and others may work differently - view the docs if you're interested.

Share Improve this answer Follow

answered Oct 27 '16 at 15:45
Lee Benson
9,817 5 40 53

2 also great for tree shaking 2ality.com/2015/12/webpack-tree-shaking.html – prosti Dec 2 '16 at 0:32

30

When it comes to async or maybe lazy loading, then `import ()` is much more powerful. See when we require the component in asynchronous way, then we use `import` it in some async manner as in `const` variable using `await` .

```
const module = await import('./module.js');
```

Or if you want to use `require()` then,

```
const converter = require('./converter');
```

Thing is `import()` is actually async in nature. As mentioned by neehar venugopal in ReactConf, you can use it to dynamically load react components for client side architecture.

Also it is way better when it comes to Routing. That is the one special thing that makes network log to download a necessary part when user connects to specific website to its specific component. e.g. login page before dashboard wouldn't download all components of dashboard. Because what is needed current i.e. login component, that only will be downloaded.

Join Stack Overflow to learn, share knowledge, and build your career.

Sign up with email Sign up with Google Sign up with GitHub Sign up with Facebook

NOTE - If you are developing a node.js project, then you have to strictly use `require()` as node will throw exception error as `invalid token 'import'` if you will use `import` . So node does not support import statements.

UPDATE - As suggested by Dan Dascalescu: Since v8.5.0 (released Sep 2017), `node --experimental-modules index.mjs` lets you use `import` without Babel. You can (and should) also publish your npm packages as native ESModule, with backwards compatibility for the old `require` way.

See this for more clearance where to use async imports - https://www.youtube.com/watch?v=bb6RCrDaxhw

Share Improve this answer Follow

edited Feb 10 '19 at 22:16
Joel
249 2 10

answered Nov 28 '17 at 2:37
Meet Zaveri
2,498 1 19 28

1 So will the require be sync and wait? – baklazan Oct 29 '18 at 13:58

1 Can say factually! – Meet Zaveri Oct 29 '18 at 14:14

17

The most important thing to know is that ES6 modules are, indeed, an official standard, while CommonJS (Node.js) modules are not.

In 2019, ES6 modules are supported by 84% of browsers. While Node.js puts them behind an --experimental-modules flag, there is also a convenient node package called esm, which makes the integration smooth.

Another issue you're likely to run into between these module systems is code location. Node.js assumes source is kept in a `node_modules` directory, while most ES6 modules are deployed in a flat directory structure. These are not easy to reconcile, but it can be done by hacking your `package.json` file with pre and post installation scripts. Here is an example isomorphic module and an article explaining how it works.

Share Improve this answer Follow

answered Apr 14 '19 at 15:25
isysd
341 2 4

10

I personally use import because, we can import the required methods, members by using import.

```
import {foo, bar} from "dep";
```

FileName: dep.js

```
export foo function(){};
export const bar = 22
```

Credit goes to Paul Shan. More info.

Share Improve this answer Follow

answered Nov 22 '17 at 7:04
chandoo
1,138 1 18 28

1 Great choice! Are you also publishing your npm packages as native ESModule, with backwards compatibility for the old `require` way? – Dan Dascalescu Sep 26 '18 at 3:37

Join Stack Overflow to learn, share knowledge, and build your career.

Sign up with email Sign up with Google Sign up with GitHub Sign up with Facebook ✕

0

Not sure why (probably optimization - lazy loading?) is it working like that, but I have noticed that `import` may not parse code if imported modules are not used.
Which may not be expected behaviour in some cases.

Take hated Foo class as our sample dependency.

foo.ts

```
export default class Foo {}
console.log('Foo loaded');
```

For example:

index.ts

```
import Foo from './foo'
// prints nothing
```

index.ts

```
const Foo = require('./foo').default;
// prints "Foo loaded"
```

index.ts

```
(async () => {
    const FooPack = await import('./foo');
    // prints "Foo loaded"
})();
```

On the other hand:

index.ts

```
import Foo from './foo'
typeof Foo; // any use case
// prints "Foo loaded"
```

Share Improve this answer Follow

edited Nov 30 '19 at 2:21

answered Nov 30 '19 at 2:14
l00k
1,377 1 15 29