ES Modules

This guide helps explain what ES Modules are and how to make a Nuxt app (or upstream library) compatible with ESM.

Background

CommonJS Modules

CommonJS (CJS) is a format introduced by Node.js that allows sharing functionality between isolated JavaScript modules (read more). You might be already familiar with this syntax:

```
const a = require('./a')

module.exports.a = a
```

Bundlers like webpack and Rollup support this syntax and allow you to use modules written in CommonJS in the browser.

ESM Syntax

Most of the time, when people talk about ESM vs CJS, they are talking about a different syntax for writing modules.

```
import a from './a'
```

```
export { a }
```

Before ECMAScript Modules (ESM) became a standard (it took more than 10 years!), tooling like webpack and even languages like TypeScript started supporting so-called **ESM syntax**. However, there are some key differences with actual spec; here's a helpful explainer.

What is 'Native' ESM?

You may have been writing your app using ESM syntax for a long time. After all, it's natively supported by the browser, and in Nuxt 2 we compiled all the code you wrote to the appropriate format (CJS for server, ESM for browser).

When using modules you'd install into your package, things were a little different. A sample library might expose both CJS and ESM versions, and let us pick which one we wanted:

```
{
  "name": "sample-library",
  "main": "dist/sample-library.cjs.js",
  "module": "dist/sample-library.esm.js"
}
```

So in Nuxt 2, the bundler (webpack) would pull in the CJS file ('main') for the server build and use the ESM file ('module') for the client build.

However, in recent Node.js LTS releases, it is now possible to use native ESM module within Node.js. That means that Node.js itself can process JavaScript using ESM syntax, although it doesn't do it by default. The two most common ways to enable ESM syntax are:

- set `type: 'module'` within your `package.json` and keep using `.js` extension
- use the `.mjs` file extensions (recommended)

This is what we do for Nuxt Nitro; we output a `.output/server/index.mjs` file. That tells Node.js to treat this file as a native ES module.

What Are Valid Imports in a Node.js Context?

When you `import` a module rather than `require` it, Node.js resolves it differently. For example, when you import `sample-library`, Node.js will look not for the `main` but for the `exports` or `module` entry in that library's `package.json`.

This is also true of dynamic imports, like `const b = await import('sample-library')` .

Node supports the following kinds of imports (see docs):

1. files ending in `.mjs` - these are expected to use ESM syntax
2. files ending in `.cjs` - these are expected to use CJS syntax
3. files ending in `.js` - these are expected to use CJS syntax unless their `package.json` has `type: 'module'`

What Kinds of Problems Can There Be?

For a long time module authors have been producing ESM-syntax builds but using conventions like `.esm.js` or `.es.js` , which they have added to the `module` field in their `package.json` . This hasn't been a problem until now because they have only been used by bundlers like webpack, which don't especially care about the file extension.

However, if you try to import a package with an `.esm.js` file in a Node.js ESM context, it won't work, and you'll get an error like:

```
> (node:22145) Warning: To load an ES module, set "type": "module" in the package.json or use the
> /path/to/index.js:1
>
> export default {}
> ^^^^^^
>
> SyntaxError: Unexpected token 'export'
>     at wrapSafe (internal/modules/cjs/loader.js:1001:16)
>     at Module._compile (internal/modules/cjs/loader.js:1049:27)
>     at Object.Module._extensions..js (internal/modules/cjs/loader.js:1114:10)
>     ....
>     at async Object.loadESM (internal/process/esm_loader.js:68:5)
```

You might also get this error if you have a named import from an ESM-syntax build that Node.js thinks is CJS:

```
> file:///path/to/index.mjs:5
> import { named } from 'sample-library'
>          ^^^^^
> SyntaxError: Named export 'named' not found. The requested module 'sample-library' is a CommonJS
>
```

```
> CommonJS modules can always be imported via the default export, for example using:
>
> import pkg from 'sample-library';
> const { named } = pkg;
>
>     at ModuleJob._instantiate (internal/modules/esm/module_job.js:120:21)
>     at async ModuleJob.run (internal/modules/esm/module_job.js:165:5)
>     at async Loader.import (internal/modules/esm/loader.js:177:24)
>     at async Object.loadESM (internal/process/esm_loader.js:68:5)
```

Troubleshooting ESM Issues

If you encounter these errors, the issue is almost certainly with the upstream library. They need to fix their library to support being imported by Node.

Transpiling Libraries

In the meantime, you can tell Nuxt not to try to import these libraries by adding them to `build.transpile` :

```
export default defineNuxtConfig({
  build: {
    transpile: ['sample-library']
  }
})
```

You may find that you *also* need to add other packages that are being imported by these libraries.

Aliasing Libraries

In some cases, you may also need to manually alias the library to the CJS version, for example:

```
export default defineNuxtConfig({
  alias: {
    'sample-library': 'sample-library/dist/sample-library.cjs.js'
  }
})
```

Default Exports

A dependency with CommonJS format, can use `module.exports` or `exports` to provide a default export:

```
node_modules/cjs-pkg/index.js

module.exports = { test: 123 }
// or
exports.test = 123
```

This normally works well if we `require` such dependency:

```
test.cjs

const pkg = require('cjs-pkg')

console.log(pkg) // { test: 123 }
```

Node.js in native ESM mode, typescript with `esModuleInterop` enabled and bundlers such as webpack, provide a compatibility mechanism so that we can default import such library. This mechanism is often referred to as "interop require default":

```
import pkg from 'cjs-pkg'

console.log(pkg) // { test: 123 }
```

However, because of the complexities of syntax detection and different bundle formats, there is always a chance that the interop default fails and we end up with something like this:

```
import pkg from 'cjs-pkg'

console.log(pkg) // { default: { test: 123 } }
```

Also when using dynamic import syntax (in both CJS and ESM files), we always have this situation:

```
import('cjs-pkg').then(console.log) // [Module: null prototype] { default: { test: '123' } }
```

In this case, we need to manually interop the default export:

```
// Static import
import { default as pkg } from 'cjs-pkg'

// Dynamic import
import('cjs-pkg').then(m => m.default || m).then(console.log)
```

For handling more complex situations and more safety, we recommend and internally use mlly in Nuxt 3 that can preserve named exports.

```
import { interopDefault } from 'mlly'

// Assuming the shape is { default: { foo: 'bar' }, baz: 'qux' }
import myModule from 'my-module'

console.log(interopDefault(myModule)) // { foo: 'bar', baz: 'qux' }
```

Library Author Guide

The good news is that it's relatively simple to fix issues of ESM compatibility. There are two main options:

1. **You can rename your ESM files to end with `.mjs`.**
   This is the recommended and simplest approach. You may have to sort out issues with your library's dependencies and possibly with your build system, but in most cases, this should fix the problem for you. It's also recommended to rename your CJS files to end with `.cjs`, for the greatest explicitness.

2. **You can opt to make your entire library ESM-only**.
   This would mean setting `type: 'module'` in your `package.json` and ensuring that your built library uses ESM syntax. However, you may face issues with your dependencies - and this approach means your library can *only* be consumed in an ESM context.

Migration

The initial step from CJS to ESM is updating any usage of `require` to use `import` instead:

Before After

```
module.exports = ...

exports.hello = ...
```

Before

Before After

```
const myLib = require('my-lib')
```

Before

In ESM Modules, unlike CJS, `require` , `require.resolve` , `__filename` and `__dirname` globals are not available and should be replaced with `import()` and `import.meta.filename` .

Before After

```
import { join } from 'path'

const newDir = join(__dirname, 'new-dir')
```

Before

Before After

```
const someFile = require.resolve('./lib/foo.js')
```

Before

Best Practices

- Prefer named exports rather than default export. This helps reduce CJS conflicts. (see Default exports section)
- Avoid depending on Node.js built-ins and CommonJS or Node.js-only dependencies as much as possible to make your library usable in Browsers and Edge Workers without needing Nitro polyfills.
- Use new `exports` field with conditional exports. (read more).

```
{
  "exports": {
    ".": {
      "import": "./dist/mymodule.mjs"
    }
  }
}
```