FEB 12, 2023

How to Create a Hybrid NPM Module for ESM and CommonJS.

How can you easily create an NPM module for ESM and CommonJS?

Preferably without using creating two source bases and without needing Webpack?

That has been a vexing question for a while.

Creating an NPM module from single code base that easily targets both CommonJS and ES modules can be an exercise in frustration. Sometimes called a “hybrid” package, easily creating an NPM module that is simply consumed using `import` or `require` is an elusive goal.

On this topic, there are countless blog articles, stack overflow questions and reference pages. These point to various competing strategies, which when tried, work in some situations but are brittle and fail in other situations. Most solutions require either Webpack, Rollup, custom scripting and build tools or creating and maintaining a dual source base. And most do not generate efficient, pure ESM code.

When reading Node documentation, you read about Webpack and Rollup, ESM, CommonJS, UMD and AMD. You read that `.mjs` and `.cjs` extensions are the solution and the future, but it seems that most developers hate them.

You read about the package.json `type = "module"` and `exports` keywords which will magically make everything work, but they don’t work as advertised.

> What a god damn mess!!

Creating a Hybrid Module should not be this difficult!

I’ve tried the `.mjs` and `.cjs` extensions which fail with more than a few essential build tools.

I’ve tried using bundlers: Webpack and Rollup.

I’ve tried the package.json `type` field, but it failed when used in combination with the package.json exports map (more below).

I’ve tried so many approaches, only to find they fail in one or more use cases.

Finally, I found a solution that is easy, works well and generates efficient ESM code. It supports a single source code base and creates a module that can be consumed by CommonJS and ESM apps and modules.

I don’t vouch that this will work in *all* use cases. But it works for all of mine, including consumption by Webpack, the serverless framework, ESM command line tools, and other ESM or CommonJS libraries.

The problem with .mjs

Before outlining the solution, let me put a sword in a few much touted techniques.

> Why not just use `.mjs` or `.cjs` extensions to indicate ESM or CommonJS code?

Node adopted these source code file extensions to indicate the type of source file. Seems logical at first glance. Extensions typically are used to describe a file type.

This works for simple, stand-alone, non-hybrid use cases. However, if you are building a hybrid module, then using `.mjs` and `.cjs` implies that you either don't have a single code base or you are using or creating custom tooling to copy the source and change the extensions and then patch your source code to use the appropriate extensions in import statements.

ESM code requires that `import` directives specify the path to the imported file. If you import from a URL with `.mjs` that code requires patching to be able to require from a `.cjs` file and vice-versa.

Further, most tool chains do not yet properly support `.mjs` files. And some web servers do not have the `.mjs` extension defined as an 'application/json' mime type. Your favorite bundler may also not understand these files. Consequently, you are writing config and mapping routines or writing custom scripts to manage these files.

I'm yet to find someone who "loves" the `.mjs` and `.cjs` extensions. Fortunately, there are alternatives. Enter the package.json `type` property.

The problem with the package.json type property

To resolve the problem of whether a file with a `.js` extension is an ES module or CommonJS module, Node invented the package.json `type` property and conventions. If you set the `type` to "module", then all files in that directory and sub-directories are considered to be ESM until either another package.json or node_modules directory is encountered. If you set the `type` to "commonjs", all files are assumed to be CommonJS.

These defaults can be overridden by explicitly naming a file with a `.cjs` or `.mjs` extension.

package.json:

```
{
    "version": "1.2.3",
    "type": "module"
}
```

This works fairly well but your package is either a "module" or "commonjs" by default. The problem is what happens when you need a package to be a hybrid and export both ESM and CommonJS formats? Unfortunately there is no way to have a conditional type that can be "module" when consumed as ESM and "commonjs" when consumed by CommonJS.

Node does provide a conditional `exports` property that defines the package's export entry points. However, this does not redefine the package type and the `type` and `exports` properties do not combine well.

The problem with package.json conditional exports

The conditional `exports` property defines a set of entry points. For our purposes, we're interested in the `import` and `require` selectors which enable a hybrid module to define different entry points for use by ESM and CommonJS.

package.json:

```
{
    "exports": {
        "import": "./dist/mjs/index.js",
        "require": "./dist/cjs/index.js"
    }
}
```

Using tooling (see below), we generate two distributions from a single source code base to target ESM and CommonJS. The `exports` property then directs Node to load the relevant entry point.

However, what happens if we have defined a package with a `type` of module and `exports` for both ESM and CommonJS. All works fine for loading the index.js, but if that file then loads another sub-module (e.g. ./submodule.js), then that file is loaded according to the package.json `type` setting and not the exports setting.

In other words, if a CommonJS app/library used this module to `require` and load from "./dist/cjs/index.js", and the 'index.js' then calls `require('./submodule.js')`, that will fail because the module's package.json had a `type` set to `module` and ESM modules prohibit the use of require.

Unfortunately, if Node loads using the exports.require, it does not assume the code below is CommonJS. It would be ideal if the exports could define a module type to override the top level package.json type.

For example, a hypothetical package.json (don't use, not supported by Node):

```
{
    "exports": {
        "import": {
            "path": "./dist/mjs/index.js",
            "type": "module"
        },
        "require": {
            "path": "./dist/cjs/index.js",
            "type": "commonjs"
        }
    }
}
```

But this is just a pipe dream.

One more wrinkle, TypeScript does not (yet) behave with exports. So you need to include the legacy `module` and `main` properties for TypeScript. The `main` property points to the CJS entry point and the `module` property points to the ESM entry.

```
"main": "dist/cjs/index.js",
"module": "dist/mjs/index.js",
```

The solution

Okay, so what is an approach that works to deliver:

- A single source code base
- Easy building
- Generates native ESM code
- Works with existing tooling
- Generates a hybrid package for either ESM or CommonJS

Single Source Base

Author your code in ES6, ES-Next or Typescript using import and export.

From this base, you can import either ES modules or CommonJS modules using import. The reverse is not true. If you author in CommonJS you cannot easily consume ES modules.

```
import Shape from './Shape.js'

export class MyShape {
    constructor() {
        this.shape = new Shape()
    }
}
```

Take care when using `export default` and then importing using `require` via CommonJS. The TypeScript or Babel transpilers will automatically bundle exports into a module.exports and then generate a ".default" reference for you when importing, however native NodeJS will not. This means if you are not using a transpiler, you may need to use a `.default` reference.

```
import Shape from './Shape.js'

const shape = new Shape.default()
```

Building

Build the source twice, once for ESM and once for CommonJS.

We use Typescript as our transpiler, and author in ES6/ES-Next or Typescript. Alternatively, Babel would work fine for ES6.

Javascript files should have a `.js` extension and not a `.mjs` or `.cjs` extension. Typescript files will have a `.ts` extension.

Here is our package.json build script:

package.json:

```
{
    "scripts": {
        "build": "rm -fr dist/* && tsc -p tsconfig.json && tsc -p tsconfig-cjs.json &&
./fixup"
    }
}
```

The `tsconfig.json` is setup to build for ESM and `tsconfig-cjs.json` builds for CommonJS.

To avoid duplication of settings, we define a shared `tsconfig-base.json` that contains shared build settings used for both ESM and CommonJS builds.

The default `tsconfig.json` is for ESM and builds using “esnext”. You can change this to “es2015” or any preset you desire.

tsconfig.json:

```
{
    "extends": "./tsconfig-base.json",
    "compilerOptions": {
        "module": "esnext",
        "outDir": "dist/mjs",
        "target": "esnext"
    }
}
```

tsconfig-cjs.json:

```
{
    "extends": "./tsconfig-base.json",
    "compilerOptions": {
        "module": "commonjs",
        "outDir": "dist/cjs",
        "target": "es2015"
    }
}
```

Here is our tsconfig-base.json for ES6 code with all shared settings:

tsconfig-base.json:

```
{
    "compilerOptions": {
        "allowJs": true,
        "allowSyntheticDefaultImports": true,
        "baseUrl": "src",
        "declaration": true,
        "esModuleInterop": true,
        "inlineSourceMap": false,
        "lib": ["esnext"],
        "listEmittedFiles": false,
        "listFiles": false,
        "moduleResolution": "node",
        "noFallthroughCasesInSwitch": true,
        "pretty": true,
        "resolveJsonModule": true,
        "rootDir": "src",
        "skipLibCheck": true,
        "strict": true,
        "traceResolution": false,
        "types": ["node", "jest"]
    },
    "compileOnSave": false,
    "exclude": ["node_modules", "dist"],
    "include": ["src"]
}
```

Per ESM/CJS package.json

The last step of the build is a simple `fixup` script that creates per-distribution `package.json` files. These package.json files define the default package type for the `.dist/*` sub-directories.

fixup:

```
cat >dist/cjs/package.json <<!EOF
{
    "type": "commonjs"
}
!EOF

cat >dist/mjs/package.json <<!EOF
{
    "type": "module"
}
!EOF
```

Package.json

Our package.json does not have a `type` property. Rather, we push that down to the package.json files under the `./dist/*` sub-directories.

We define an `exports` map which defines the entry points for the package: one for ESM and one for CJS. Read more in the Node Documentation about conditional exports.

Here is a segment of our package.json:

package.json:

```
"main": "dist/cjs/index.js",
"module": "dist/mjs/index.js",
"exports": {
    ".": {
        "import": "./dist/mjs/index.js",
        "require": "./dist/cjs/index.js"
    }
},
```

Summary

With the above strategy, modules can be consumed using `import` or `require`. And you can use a single code base that uses modern ES6 or Typescript. Users of your ESM distribution get the benefit of increased performance and easier debugging.

We use the approach above for our NPM modules. See the following modules for examples:

- DynamoDB OneTable
- OneTable Migrate
- OneTable

To learn more about SenseDeep and our Serverless Developer Studio, please visit https://www.sensedeep.com/.

Links

- SenseDeep App
- GitHub OneTable
- NPM OneTable