Modules: CommonJS modules

Stability: 2 - Stable

CommonJS modules are the original way to package JavaScript code for Node.js. Node.js also supports the ECMAScript modules standard used by browsers and other JavaScript runtimes.

In Node.js, each file is treated as a separate module. For example, consider a file named `foo.js`:

```
const circle = require('./circle.js');
console.log(`The area of a circle of radius 4 is ${circle.area(4)}`);
```

On the first line, `foo.js` loads the module `circle.js` that is in the same directory as `foo.js`.

Here are the contents of `circle.js`:

```
const { PI } = Math;

exports.area = (r) => PI * r ** 2;

exports.circumference = (r) => 2 * PI * r;
```

The module `circle.js` has exported the functions `area()` and `circumference()`. Functions and objects are added to the root of a module by specifying additional properties on the special `exports` object.

Variables local to the module will be private, because the module is wrapped in a function by Node.js (see module wrapper). In this example, the variable `PI` is private to `circle.js`.

The `module.exports` property can be assigned a new value (such as a function or object).

Below, `bar.js` makes use of the `square` module, which exports a Square class:

```
const Square = require('./square.js');
const mySquare = new Square(2);
console.log(`The area of mySquare is ${mySquare.area()}`);
```

The `square` module is defined in `square.js`:

```
// Assigning to exports will not modify module, must use module.exports
module.exports = class Square {
  constructor(width) {
    this.width = width;
  }

  area() {
    return this.width ** 2;
```

```
  }
};
```

The CommonJS module system is implemented in the `module` core module.

Enabling

Node.js has two module systems: CommonJS modules and ECMAScript modules.

By default, Node.js will treat the following as CommonJS modules:

- Files with a `.cjs` extension;
- Files with a `.js` extension when the nearest parent `package.json` file contains a top-level field `"type"` with a value of `"commonjs"`.
- Files with a `.js` extension when the nearest parent `package.json` file doesn't contain a top-level field `"type"`. Package authors should include the `"type"` field, even in packages where all sources are CommonJS. Being explicit about the `type` of the package will make things easier for build tools and loaders to determine how the files in the package should be interpreted.
- Files with an extension that is not `.mjs`, `.cjs`, `.json`, `.node`, or `.js` (when the nearest parent `package.json` file contains a top-level field `"type"` with a value of `"module"`, those files will be recognized as CommonJS modules only if they are being `require`d, not when used as the command-line entry point of the program).

See Determining module system for more details.

Calling `require()` always use the CommonJS module loader. Calling `import()` always use the ECMAScript module loader.

Accessing the main module

When a file is run directly from Node.js, `require.main` is set to its `module`. That means that it is possible to determine whether a file has been run directly by testing `require.main === module`.

For a file `foo.js`, this will be `true` if run via `node foo.js`, but `false` if run by `require('./foo')`.

When the entry point is not a CommonJS module, `require.main` is `undefined`, and the main module is out of reach.

Package manager tips

The semantics of the Node.js `require()` function were designed to be general enough to support reasonable directory structures. Package manager programs

such as `dpkg`, `rpm`, and `npm` will hopefully find it possible to build native packages from Node.js modules without modification.

Below we give a suggested directory structure that could work:

Let's say that we wanted to have the folder at `/usr/lib/node/<some-package>/<some-version>` hold the contents of a specific version of a package.

Packages can depend on one another. In order to install package `foo`, it may be necessary to install a specific version of package `bar`. The `bar` package may itself have dependencies, and in some cases, these may even collide or form cyclic dependencies.

Because Node.js looks up the `realpath` of any modules it loads (that is, it resolves symlinks) and then looks for their dependencies in `node_modules` folders, this situation can be resolved with the following architecture:

- `/usr/lib/node/foo/1.2.3/`: Contents of the `foo` package, version 1.2.3.
- `/usr/lib/node/bar/4.3.2/`: Contents of the `bar` package that `foo` depends on.
- `/usr/lib/node/foo/1.2.3/node_modules/bar`: Symbolic link to `/usr/lib/node/bar/4.3.2/`.
- `/usr/lib/node/bar/4.3.2/node_modules/*`: Symbolic links to the packages that `bar` depends on.

Thus, even if a cycle is encountered, or if there are dependency conflicts, every module will be able to get a version of its dependency that it can use.

When the code in the `foo` package does `require('bar')`, it will get the version that is symlinked into `/usr/lib/node/foo/1.2.3/node_modules/bar`. Then, when the code in the `bar` package calls `require('quux')`, it'll get the version that is symlinked into `/usr/lib/node/bar/4.3.2/node_modules/quux`.

Furthermore, to make the module lookup process even more optimal, rather than putting packages directly in `/usr/lib/node`, we could put them in `/usr/lib/node_modules/<name>/<version>`. Then Node.js will not bother looking for missing dependencies in `/usr/node_modules` or `/node_modules`.

In order to make modules available to the Node.js REPL, it might be useful to also add the `/usr/lib/node_modules` folder to the `$NODE_PATH` environment variable. Since the module lookups using `node_modules` folders are all relative, and based on the real path of the files making the calls to `require()`, the packages themselves can be anywhere.

The `.mjs` extension

Due to the synchronous nature of `require()`, it is not possible to use it to load ECMAScript module files. Attempting to do so will throw a `ERR_REQUIRE_ESM` error. Use `import()` instead.

The `.mjs` extension is reserved for ECMAScript Modules which cannot be loaded via `require()`. See Determining module system section for more info regarding which files are parsed as ECMAScript modules.

All together

To get the exact filename that will be loaded when `require()` is called, use the `require.resolve()` function.

Putting together all of the above, here is the high-level algorithm in pseudocode of what `require()` does:

Caching

Modules are cached after the first time they are loaded. This means (among other things) that every call to `require('foo')` will get exactly the same object returned, if it would resolve to the same file.

Provided `require.cache` is not modified, multiple calls to `require('foo')` will not cause the module code to be executed multiple times. This is an important feature. With it, "partially done" objects can be returned, thus allowing transitive dependencies to be loaded even when they would cause cycles.

To have a module execute code multiple times, export a function, and call that function.

Module caching caveats

Modules are cached based on their resolved filename. Since modules may resolve to a different filename based on the location of the calling module (loading from `node_modules` folders), it is not a *guarantee* that `require('foo')` will always return the exact same object, if it would resolve to different files.

Additionally, on case-insensitive file systems or operating systems, different resolved filenames can point to the same file, but the cache will still treat them as different modules and will reload the file multiple times. For example, `require('./foo')` and `require('./FOO')` return two different objects, irrespective of whether or not `./foo` and `./FOO` are the same file.

Core modules

Node.js has several modules compiled into the binary. These modules are described in greater detail elsewhere in this documentation.

The core modules are defined within the Node.js source and are located in the `lib/` folder.

Core modules are always preferentially loaded if their identifier is passed to `require()`. For instance, `require('http')` will always return the built in

HTTP module, even if there is a file by that name.

Core modules can also be identified using the `node:` prefix, in which case it bypasses the `require` cache. For instance, `require('node:http')` will always return the built in HTTP module, even if there is `require.cache` entry by that name.

Cycles

When there are circular `require()` calls, a module might not have finished executing when it is returned.

Consider this situation:

`a.js`:

```
console.log('a starting');
exports.done = false;
const b = require('./b.js');
console.log('in a, b.done = %j', b.done);
exports.done = true;
console.log('a done');
```

`b.js`:

```
console.log('b starting');
exports.done = false;
const a = require('./a.js');
console.log('in b, a.done = %j', a.done);
exports.done = true;
console.log('b done');
```

`main.js`:

```
console.log('main starting');
const a = require('./a.js');
const b = require('./b.js');
console.log('in main, a.done = %j, b.done = %j', a.done, b.done);
```

When `main.js` loads `a.js`, then `a.js` in turn loads `b.js`. At that point, `b.js` tries to load `a.js`. In order to prevent an infinite loop, an **unfinished copy** of the `a.js` exports object is returned to the `b.js` module. `b.js` then finishes loading, and its `exports` object is provided to the `a.js` module.

By the time `main.js` has loaded both modules, they're both finished. The output of this program would thus be:

```
$ node main.js
main starting
a starting
b starting
```

```
in b, a.done = false
b done
in a, b.done = true
a done
in main, a.done = true, b.done = true
```

Careful planning is required to allow cyclic module dependencies to work correctly within an application.

File modules

If the exact filename is not found, then Node.js will attempt to load the required filename with the added extensions: `.js`, `.json`, and finally `.node`. When loading a file that has a different extension (e.g. `.cjs`), its full name must be passed to `require()`, including its file extension (e.g. `require('./file.cjs')`).

`.json` files are parsed as JSON text files, `.node` files are interpreted as compiled addon modules loaded with `process.dlopen()`. Files using any other extension (or no extension at all) are parsed as JavaScript text files. Refer to the Determining module system section to understand what parse goal will be used.

A required module prefixed with `'/'` is an absolute path to the file. For example, `require('/home/marco/foo.js')` will load the file at `/home/marco/foo.js`.

A required module prefixed with `'./'` is relative to the file calling `require()`. That is, `circle.js` must be in the same directory as `foo.js` for `require('./circle')` to find it.

Without a leading `'/'`, `'./'`, or `'../'` to indicate a file, the module must either be a core module or is loaded from a `node_modules` folder.

If the given path does not exist, `require()` will throw a `MODULE_NOT_FOUND` error.

Folders as modules

> Stability: 3 - Legacy: Use subpath exports or subpath imports instead.

There are three ways in which a folder may be passed to `require()` as an argument.

The first is to create a `package.json` file in the root of the folder, which specifies a `main` module. An example `package.json` file might look like this:

```
{ "name" : "some-library",
  "main" : "./lib/some-library.js" }
```

If this was in a folder at `./some-library`, then `require('./some-library')` would attempt to load `./some-library/lib/some-library.js`.

If there is no `package.json` file present in the directory, or if the `"main"` entry is missing or cannot be resolved, then Node.js will attempt to load an `index.js` or `index.node` file out of that directory. For example, if there was no `package.json` file in the previous example, then `require('./some-library')` would attempt to load:

- `./some-library/index.js`
- `./some-library/index.node`

If these attempts fail, then Node.js will report the entire module as missing with the default error:

```
Error: Cannot find module 'some-library'
```

In all three above cases, an `import('./some-library')` call would result in a `ERR_UNSUPPORTED_DIR_IMPORT` error. Using package subpath exports or subpath imports can provide the same containment organization benefits as folders as modules, and work for both `require` and `import`.

Loading from `node_modules` folders

If the module identifier passed to `require()` is not a core module, and does not begin with `'/'`, `'../'`, or `'./'`, then Node.js starts at the directory of the current module, and adds `/node_modules`, and attempts to load the module from that location. Node.js will not append `node_modules` to a path already ending in `node_modules`.

If it is not found there, then it moves to the parent directory, and so on, until the root of the file system is reached.

For example, if the file at `'/home/ry/projects/foo.js'` called `require('bar.js')`, then Node.js would look in the following locations, in this order:

- `/home/ry/projects/node_modules/bar.js`
- `/home/ry/node_modules/bar.js`
- `/home/node_modules/bar.js`
- `/node_modules/bar.js`

This allows programs to localize their dependencies, so that they do not clash.

It is possible to require specific files or sub modules distributed with a module by including a path suffix after the module name. For instance `require('example-module/path/to/file')` would resolve `path/to/file` relative to where `example-module` is located. The suffixed path follows the same module resolution semantics.

Loading from the global folders

If the `NODE_PATH` environment variable is set to a colon-delimited list of absolute paths, then Node.js will search those paths for modules if they are not found

elsewhere.

On Windows, `NODE_PATH` is delimited by semicolons (`;`) instead of colons.

`NODE_PATH` was originally created to support loading modules from varying paths before the current module resolution algorithm was defined.

`NODE_PATH` is still supported, but is less necessary now that the Node.js ecosystem has settled on a convention for locating dependent modules. Sometimes deployments that rely on `NODE_PATH` show surprising behavior when people are unaware that `NODE_PATH` must be set. Sometimes a module's dependencies change, causing a different version (or even a different module) to be loaded as the `NODE_PATH` is searched.

Additionally, Node.js will search in the following list of GLOBAL_FOLDERS:

- 1: `$HOME/.node_modules`
- 2: `$HOME/.node_libraries`
- 3: `$PREFIX/lib/node`

Where `$HOME` is the user's home directory, and `$PREFIX` is the Node.js configured `node_prefix`.

These are mostly for historic reasons.

It is strongly encouraged to place dependencies in the local `node_modules` folder. These will be loaded faster, and more reliably.

The module wrapper

Before a module's code is executed, Node.js will wrap it with a function wrapper that looks like the following:

```
(function(exports, require, module, __filename, __dirname) {
// Module code actually lives in here
});
```

By doing this, Node.js achieves a few things:

- It keeps top-level variables (defined with `var`, `const` or `let`) scoped to the module rather than the global object.
- It helps to provide some global-looking variables that are actually specific to the module, such as:
  - The `module` and `exports` objects that the implementor can use to export values from the module.
  - The convenience variables `__filename` and `__dirname`, containing the module's absolute filename and directory path.

The module scope

`__dirname`

- {string}

The directory name of the current module. This is the same as the `path.dirname()` of the `__filename`.

Example: running `node example.js` from `/Users/mjr`

```
console.log(__dirname);
// Prints: /Users/mjr
console.log(path.dirname(__filename));
// Prints: /Users/mjr
```

`__filename`

- {string}

The file name of the current module. This is the current module file's absolute path with symlinks resolved.

For a main program this is not necessarily the same as the file name used in the command line.

See `__dirname` for the directory name of the current module.

Examples:

Running `node example.js` from `/Users/mjr`

```
console.log(__filename);
// Prints: /Users/mjr/example.js
console.log(__dirname);
// Prints: /Users/mjr
```

Given two modules: `a` and `b`, where `b` is a dependency of `a` and there is a directory structure of:

- `/Users/mjr/app/a.js`
- `/Users/mjr/app/node_modules/b/b.js`

References to `__filename` within `b.js` will return `/Users/mjr/app/node_modules/b/b.js` while references to `__filename` within `a.js` will return `/Users/mjr/app/a.js`.

`exports`

- {Object}

A reference to the `module.exports` that is shorter to type. See the section about the exports shortcut for details on when to use `exports` and when to use `module.exports`.

`module`

- {module}

A reference to the current module, see the section about the `module` object. In particular, `module.exports` is used for defining what a module exports and makes available through `require()`.

`require(id)`

- `id` {string} module name or path
- Returns: {any} exported module content

Used to import modules, `JSON`, and local files. Modules can be imported from `node_modules`. Local modules and JSON files can be imported using a relative path (e.g. `./`, `./foo`, `./bar/baz`, `../foo`) that will be resolved against the directory named by `__dirname` (if defined) or the current working directory. The relative paths of POSIX style are resolved in an OS independent fashion, meaning that the examples above will work on Windows in the same way they would on Unix systems.

```
// Importing a local module with a path relative to the `__dirname` or current
// working directory. (On Windows, this would resolve to .\path\myLocalModule.)
const myLocalModule = require('./path/myLocalModule');

// Importing a JSON file:
const jsonData = require('./path/filename.json');

// Importing a module from node_modules or Node.js built-in module:
const crypto = require('crypto');
```

`require.cache`

- {Object}

Modules are cached in this object when they are required. By deleting a key value from this object, the next `require` will reload the module. This does not apply to native addons, for which reloading will result in an error.

Adding or replacing entries is also possible. This cache is checked before native modules and if a name matching a native module is added to the cache, only `node:`-prefixed require calls are going to receive the native module. Use with care!

```
const assert = require('assert');
const realFs = require('fs');

const fakeFs = {};
require.cache.fs = { exports: fakeFs };
```

```
assert.strictEqual(require('fs'), fakeFs);
assert.strictEqual(require('node:fs'), realFs);
```

`require.extensions`

Stability: 0 - Deprecated

- {Object}

Instruct `require` on how to handle certain file extensions.

Process files with the extension `.sjs` as `.js`:

```
require.extensions['.sjs'] = require.extensions['.js'];
```

Deprecated. In the past, this list has been used to load non-JavaScript modules into Node.js by compiling them on-demand. However, in practice, there are much better ways to do this, such as loading modules via some other Node.js program, or compiling them to JavaScript ahead of time.

Avoid using `require.extensions`. Use could cause subtle bugs and resolving the extensions gets slower with each registered extension.

`require.main`

- {module | undefined}

The `Module` object representing the entry script loaded when the Node.js process launched, or `undefined` if the entry point of the program is not a CommonJS module. See "Accessing the main module".

In `entry.js` script:

```
console.log(require.main);
```

```
node entry.js
```

```
Module {
  id: '.',
  path: '/absolute/path/to',
  exports: {},
  filename: '/absolute/path/to/entry.js',
  loaded: false,
  children: [],
  paths:
   [ '/absolute/path/to/node_modules',
     '/absolute/path/node_modules',
     '/absolute/node_modules',
     '/node_modules' ] }
```

`require.resolve(request[, options])`

- `request` {string} The module path to resolve.
- `options` {Object}
  - `paths` {string[]} Paths to resolve module location from. If present, these paths are used instead of the default resolution paths, with the exception of GLOBAL_FOLDERS like `$HOME/.node_modules`, which are always included. Each of these paths is used as a starting point for the module resolution algorithm, meaning that the `node_modules` hierarchy is checked from this location.
- Returns: {string}

Use the internal `require()` machinery to look up the location of a module, but rather than loading the module, just return the resolved filename.

If the module can not be found, a `MODULE_NOT_FOUND` error is thrown.

`require.resolve.paths(request)`

- `request` {string} The module path whose lookup paths are being retrieved.
- Returns: {string[]|null}

Returns an array containing the paths searched during resolution of `request` or `null` if the `request` string references a core module, for example `http` or `fs`.

The `module` object

- {Object}

In each module, the `module` free variable is a reference to the object representing the current module. For convenience, `module.exports` is also accessible via the `exports` module-global. `module` is not actually a global but rather local to each module.

`module.children`

- {module[]}

The module objects required for the first time by this one.

`module.exports`

- {Object}

The `module.exports` object is created by the `Module` system. Sometimes this is not acceptable; many want their module to be an instance of some class. To do this, assign the desired export object to `module.exports`. Assigning the desired object to `exports` will simply rebind the local `exports` variable, which is probably not what is desired.

For example, suppose we were making a module called `a.js`:

```
const EventEmitter = require('events');

module.exports = new EventEmitter();

// Do some work, and after some time emit
// the 'ready' event from the module itself.
setTimeout(() => {
  module.exports.emit('ready');
}, 1000);
```

Then in another file we could do:

```
const a = require('./a');
a.on('ready', () => {
  console.log('module "a" is ready');
});
```

Assignment to `module.exports` must be done immediately. It cannot be done in any callbacks. This does not work:

`x.js`:

```
setTimeout(() => {
  module.exports = { a: 'hello' };
}, 0);
```

`y.js`:

```
const x = require('./x');
console.log(x.a);
```

`exports` shortcut The `exports` variable is available within a module's file-level scope, and is assigned the value of `module.exports` before the module is evaluated.

It allows a shortcut, so that `module.exports.f = ...` can be written more succinctly as `exports.f = ...`. However, be aware that like any variable, if a new value is assigned to `exports`, it is no longer bound to `module.exports`:

```
module.exports.hello = true; // Exported from require of module
exports = { hello: false };  // Not exported, only available in the module
```

When the `module.exports` property is being completely replaced by a new object, it is common to also reassign `exports`:

```
module.exports = exports = function Constructor() {
  // ... etc.
};
```

To illustrate the behavior, imagine this hypothetical implementation of `require()`, which is quite similar to what is actually done by `require()`:

```
function require(/* ... */) {
  const module = { exports: {} };
  ((module, exports) => {
    // Module code here. In this example, define a function.
    function someFunc() {}
    exports = someFunc;
    // At this point, exports is no longer a shortcut to module.exports, and
    // this module will still export an empty default object.
    module.exports = someFunc;
    // At this point, the module will now export someFunc, instead of the
    // default object.
  })(module, module.exports);
  return module.exports;
}
```

`module.filename`

- {string}

The fully resolved filename of the module.

`module.id`

- {string}

The identifier for the module. Typically this is the fully resolved filename.

`module.isPreloading`

- Type: {boolean} `true` if the module is running during the Node.js preload phase.

`module.loaded`

- {boolean}

Whether or not the module is done loading, or is in the process of loading.

`module.parent`

> Stability: 0 - Deprecated: Please use `require.main` and `module.children` instead.

- {module | null | undefined}

The module that first required this one, or `null` if the current module is the entry point of the current process, or `undefined` if the module was loaded by something that is not a CommonJS module (E.G.: REPL or `import`).

`module.path`

- {string}

The directory name of the module. This is usually the same as the `path.dirname()` of the `module.id`.

`module.paths`

- {string[]}

The search paths for the module.

`module.require(id)`

- `id` {string}
- Returns: {any} exported module content

The `module.require()` method provides a way to load a module as if `require()` was called from the original module.

In order to do this, it is necessary to get a reference to the `module` object. Since `require()` returns the `module.exports`, and the `module` is typically *only* available within a specific module's code, it must be explicitly exported in order to be used.

The `Module` object

This section was moved to Modules: `module` core module.

- `module.builtinModules`
- `module.createRequire(filename)`
- `module.syncBuiltinESMExports()`

Source map v3 support

This section was moved to Modules: `module` core module.

- `module.findSourceMap(path)`
- Class: `module.SourceMap`
  - `new SourceMap(payload)`
  - `sourceMap.payload`
  - `sourceMap.findEntry(lineNumber, columnNumber)`