Node.js v10.5.0 Documentation

Table of Contents

- Modules
  - Accessing the main module
  - Addenda: Package Manager Tips
  - All Together...
  - Caching
    - Module Caching Caveats
  - Core Modules
  - Cycles
  - File Modules
  - Folders as Modules
  - Loading from `node_modules` Folders
  - Loading from the global folders
  - The module wrapper
  - The module scope
    - __dirname
    - __filename
    - exports
    - module
    - require()
      - require.cache
      - require.extensions deprecated
      - require.main
      - require.resolve(request[, options])
      - require.resolve.paths(request)
  - The `module` Object
    - module.children
    - module.exports
      - exports shortcut
    - module.filename
    - module.id
    - module.loaded
    - module.parent
    - module.paths
    - module.require(id)
  - The `Module` Object
    - module.builtinModules

Modules

Stability: 2 - Stable

In the Node.js module system, 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 module system is implemented in the `require('module')` module.

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')`.

Because `module` provides a `filename` property (normally equivalent to `__filename`), the entry point of the current application can be obtained by checking `require.main.filename`.

Addenda: Package Manager Tips

The semantics of Node.js's `require()` function were designed to be general enough to support a number of 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.

Since Node.js looks up the `realpath` of any modules it loads (that is, resolves symlinks), and then looks for their dependencies in the `node_modules` folders as described here, this situation is very simple to resolve 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.

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.resolve()` does:

```
require(X) from module at path Y
1. If X is a core module,
   a. return the core module
   b. STOP
2. If X begins with '/'
   a. set Y to be the filesystem root
3. If X begins with './' or '/' or '../'
   a. LOAD_AS_FILE(Y + X)
   b. LOAD_AS_DIRECTORY(Y + X)
4. LOAD_NODE_MODULES(X, dirname(Y))
5. THROW "not found"

LOAD_AS_FILE(X)
1. If X is a file, load X as JavaScript text.  STOP
2. If X.js is a file, load X.js as JavaScript text.  STOP
3. If X.json is a file, parse X.json to a JavaScript Object.  STOP
4. If X.node is a file, load X.node as binary addon.  STOP

LOAD_INDEX(X)
1. If X/index.js is a file, load X/index.js as JavaScript text.  STOP
2. If X/index.json is a file, parse X/index.json to a JavaScript object. STOP
3. If X/index.node is a file, load X/index.node as binary addon.  STOP

LOAD_AS_DIRECTORY(X)
1. If X/package.json is a file,
   a. Parse X/package.json, and look for "main" field.
   b. let M = X + (json main field)
   c. LOAD_AS_FILE(M)
   d. LOAD_INDEX(M)
2. LOAD_INDEX(X)

LOAD_NODE_MODULES(X, START)
1. let DIRS = NODE_MODULES_PATHS(START)
2. for each DIR in DIRS:
   a. LOAD_AS_FILE(DIR/X)
   b. LOAD_AS_DIRECTORY(DIR/X)

NODE_MODULES_PATHS(START)
1. let PARTS = path split(START)
2. let I = count of PARTS - 1
3. let DIRS = [GLOBAL_FOLDERS]
4. while I >= 0,
   a. if PARTS[I] = "node_modules" CONTINUE
   b. DIR = path join(PARTS[0 .. I] + "node_modules")
   c. DIRS = DIRS + DIR
   d. let I = I - 1
5. return DIRS
```

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.

Multiple calls to `require('foo')` may 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 Node.js's 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.

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`.

`.js` files are interpreted as JavaScript text files, and `.json` files are parsed as JSON text files. `.node` files are interpreted as compiled addon modules loaded with `dlopen`.

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 an `Error` with its `code` property set to `'MODULE_NOT_FOUND'`.

Folders as Modules

It is convenient to organize programs and libraries into self-contained directories, and then provide a single entry point to that library. 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`.

This is the extent of Node.js's awareness of `package.json` files.

If the file specified by the `'main'` entry of `package.json` is missing and can not be resolved, Node.js will report the entire module as missing with the default error:

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

If there is no `package.json` file present in the directory, 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 above example, then `require('./some-library')` would attempt to load:

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

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 parent directory of the current module, and adds `/node_modules`, and attempts to load the module from that location. Node 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 frozen.

`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 Node.js's 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

Added in: v0.1.27

- `<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

Added in: v0.0.1

- `<string>`

The file name of the current module. This is the resolved absolute path of the current module file.

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

Added in: v0.1.12

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

Added in: v0.1.16

- `<Object>`

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()

Added in: v0.1.13

- `<Function>`

To require modules.

require.cache

Added in: v0.3.0

- `<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. Note that this does not apply to native addons, for which reloading will result in an error.

require.extensions

Added in: v0.3.0 Deprecated since: v0.10.6

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.

Since the module system is locked, this feature will probably never go away. However, it may have subtle bugs and complexities that are best left untouched.

Note that the number of file system operations that the module system has to perform in order to resolve a `require(...)` statement to a filename scales linearly with the number of registered extensions.

In other words, adding extensions slows down the module loader and should be discouraged.

require.main

Added in: v0.1.17

- `<Object>`

The `Module` object representing the entry script loaded when the Node.js process launched. See "Accessing the main module".

In `entry.js` script:

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

```
node entry.js
```

```
Module {
  id: '.',
  exports: {},
  parent: null,
  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])

▸ History

- `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. Note that 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.

require.resolve.paths(request)

Added in: v8.9.0

- `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

Added in: v0.1.16

- `<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

Added in: v0.1.16

- `<module[]>`

The module objects required by this one.

module.exports

Added in: v0.1.16

- `<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`. Note that 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');
});
```

Note that 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

Added in: v0.1.16

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

Added in: v0.1.16

- `<string>`

The fully resolved filename to the module.

module.id

Added in: v0.1.16

- `<string>`

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

module.loaded

Added in: v0.1.16

- `<boolean>`

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

module.parent

Added in: v0.1.16

- `<module>`

The module that first required this one.

module.paths

Added in: v0.4.0

- `<string[]>`

The search paths for the module.

module.require(id)

Added in: v0.5.1

- `id` `<string>`
- Returns: `<Object>` `module.exports` from the resolved module

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

Added in: v0.3.7

- `<Object>`

Provides general utility methods when interacting with instances of `Module` — the `module` variable often seen in file modules. Accessed via `require('module')`.

module.builtinModules

Added in: v9.3.0

- `<string[]>`

A list of the names of all modules provided by Node.js. Can be used to verify if a module is maintained by a third party or not.

Note that `module` in this context isn't the same object that's provided by the module wrapper. To access it, require the `Module` module:

```
const builtin = require('module').builtinModules;
```