Node.js v0.10.35 Manual & Documentation

**[Index](file:///C:\\Users\\Apple\\AppData\\Roaming\\npm\\node_modules\\learnyounode\\node_apidoc\\index.html)** | [**View on single page**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\all.html) | [**View as JSON**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.json)

Table of Contents

* [**Modules**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_modules)
  + [**Cycles**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_cycles)
  + [**Core Modules**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_core_modules)
  + [**File Modules**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_file_modules)
  + [**Loading from node\_modules Folders**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_loading_from_node_modules_folders)
  + [**Folders as Modules**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_folders_as_modules)
  + [**Caching**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_caching)
    - [**Module Caching Caveats**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_module_caching_caveats)
  + [**The module Object**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_the_module_object)
    - [**module.exports**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_module_exports)
      * [**exports alias**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_exports_alias)
    - [**module.require(id)**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_module_require_id)
    - [**module.id**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_module_id)
    - [**module.filename**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_module_filename)
    - [**module.loaded**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_module_loaded)
    - [**module.parent**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_module_parent)
    - [**module.children**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_module_children)
  + [**All Together...**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_all_together)
  + [**Loading from the global folders**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_loading_from_the_global_folders)
  + [**Accessing the main module**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_accessing_the_main_module)
  + [**Addenda: Package Manager Tips**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_addenda_package_manager_tips)

Modules[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_modules)

Stability: 5 - Locked

Node has a simple module loading system. In Node, files and modules are in one-to-one correspondence. As an example, foo.js loads the module circle.js in the same directory.

The contents of foo.js:

var circle = require('./circle.js');

console.log( 'The area of a circle of radius 4 is '

+ circle.area(4));

The contents of circle.js:

var PI = Math.PI;

exports.area = function (r) {

return PI \* r \* r;

};

exports.circumference = function (r) {

return 2 \* PI \* r;

};

The module circle.js has exported the functions area() and circumference(). To add functions and objects to the root of your module, you can add them to the special exports object.

Variables local to the module will be private, as though the module was wrapped in a function. In this example the variable PI is private to circle.js.

If you want the root of your module's export to be a function (such as a constructor) or if you want to export a complete object in one assignment instead of building it one property at a time, assign it to module.exports instead of exports.

Below, bar.js makes use of the square module, which exports a constructor:

var square = require('./square.js');

var mySquare = square(2);

console.log('The area of my square is ' + mySquare.area());

The square module is defined in square.js:

// assigning to exports will not modify module, must use module.exports

module.exports = function(width) {

return {

area: function() {

return width \* width;

}

};

}

The module system is implemented in the require("module") module.

Cycles[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_cycles)

When there are circular require() calls, a module might not be done being executed when it is returned.

Consider this situation:

a.js:

console.log('a starting');

exports.done = false;

var 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;

var 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');

var a = require('./a.js');

var 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

If you have cyclic module dependencies in your program, make sure to plan accordingly.

Core Modules[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_core_modules)

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

The core modules are defined in node's source 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.

File Modules[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_file_modules)

If the exact filename is not found, then node will attempt to load the required filename with the added extension of .js, .json, and then .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 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 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 is either 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'.

Loading from node\_modules Folders[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_loading_from_node_modules_folders)

If the module identifier passed to require() is not a native module, and does not begin with '/', '../', or './', then node starts at the parent directory of the current module, and adds /node\_modules, and attempts to load the module from that location.

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 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.

Folders as Modules[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_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's awareness of package.json files.

If there is no package.json file present in the directory, then node 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

Caching[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_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.

If you want to have a module execute code multiple times, then export a function, and call that function.

Module Caching Caveats[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_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.

The module Object[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_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 isn't actually a global but rather local to each module.

module.exports**[#](file:///C:\\Users\\Apple\\AppData\\Roaming\\npm\\node_modules\\learnyounode\\node_apidoc\\modules.html" \l "modules_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. Note that assigning the desired object to exports will simply rebind the local exports variable, which is probably not what you want to do.

For example suppose we were making a module called a.js

var EventEmitter = require('events').EventEmitter;

module.exports = new EventEmitter();

// Do some work, and after some time emit

// the 'ready' event from the module itself.

setTimeout(function() {

module.exports.emit('ready');

}, 1000);

Then in another file we could do

var a = require('./a');

a.on('ready', function() {

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(function() {

module.exports = { a: "hello" };

}, 0);

y.js:

var x = require('./x');

console.log(x.a);

exports alias[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_exports_alias)

The exports variable that is available within a module starts as a reference to module.exports. As with any variable, if you assign a new value to it, it is no longer bound to the previous value.

To illustrate the behaviour, imagine this hypothetical implementation of require():

function require(...) {

// ...

function (module, exports) {

// Your module code here

exports = some\_func; // re-assigns exports, exports is no longer

// a shortcut, and nothing is exported.

module.exports = some\_func; // makes your module export 0

} (module, module.exports);

return module;

}

As a guideline, if the relationship between exports and module.exports seems like magic to you, ignore exports and only use module.exports.

module.require(id)[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_module_require_id)

* id String
* Return: 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.

Note that in order to do this, you must 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.

module.id[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_module_id)

* String

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

module.filename**[#](file:///C:\\Users\\Apple\\AppData\\Roaming\\npm\\node_modules\\learnyounode\\node_apidoc\\modules.html" \l "modules_module_filename)**

* String

The fully resolved filename to the module.

module.loaded**[#](file:///C:\\Users\\Apple\\AppData\\Roaming\\npm\\node_modules\\learnyounode\\node_apidoc\\modules.html" \l "modules_module_loaded)**

* Boolean

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

module.parent**[#](file:///C:\\Users\\Apple\\AppData\\Roaming\\npm\\node_modules\\learnyounode\\node_apidoc\\modules.html" \l "modules_module_parent)**

* Module Object

The module that required this one.

module.children**[#](file:///C:\\Users\\Apple\\AppData\\Roaming\\npm\\node_modules\\learnyounode\\node_apidoc\\modules.html" \l "modules_module_children)**

* Array

The module objects required by this one.

All Together...[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#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.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 './' or '/' or '../'

a. LOAD\_AS\_FILE(Y + X)

b. LOAD\_AS\_DIRECTORY(Y + X)

3. LOAD\_NODE\_MODULES(X, dirname(Y))

4. 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\_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)

2. If X/index.js is a file, load X/index.js as JavaScript text. STOP

3. If X/index.json is a file, parse X/index.json to a JavaScript object. STOP

4. If X/index.node is a file, load X/index.node as binary addon. STOP

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 = []

4. while I >= 0,

a. if PARTS[I] = "node\_modules" CONTINUE

c. DIR = path join(PARTS[0 .. I] + "node\_modules")

b. DIRS = DIRS + DIR

c. let I = I - 1

5. return DIRS

Loading from the global folders[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_loading_from_the_global_folders)

If the NODE\_PATH environment variable is set to a colon-delimited list of absolute paths, then node will search those paths for modules if they are not found elsewhere. (Note: On Windows, NODE\_PATH is delimited by semicolons instead of colons.)

Additionally, node will search in the following locations:

* 1: $HOME/.node\_modules
* 2: $HOME/.node\_libraries
* 3: $PREFIX/lib/node

Where $HOME is the user's home directory, and $PREFIX is node's configured node\_prefix.

These are mostly for historic reasons. You are highly encouraged to place your dependencies locally in node\_modulesfolders. They will be loaded faster, and more reliably.

Accessing the main module[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_accessing_the_main_module)

When a file is run directly from Node, require.main is set to its module. That means that you can 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[**#**](file:///C:\Users\Apple\AppData\Roaming\npm\node_modules\learnyounode\node_apidoc\modules.html#modules_addenda_package_manager_tips)

The semantics of Node's require() function were designed to be general enough to support a number of sane directory structures. Package manager programs such as dpkg, rpm, and npm will hopefully find it possible to build native packages from Node 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, you may have to install a specific version of package bar. The bar package may itself have dependencies, and in some cases, these dependencies may even collide or form cycles.

Since Node 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 above, 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 will not bother looking for missing dependencies in /usr/node\_modules or /node\_modules.

In order to make modules available to the node 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 Node.js Project is Sponsored by](http://www.joyent.com)**

**[Production Node +  
High Performance Infrastructure](http://www.joyent.com)**

[Get Started](https://my.joyent.com/landing/signup/701800000015696)

* [Downloads](http://nodejs.org/download/)
* [Documentation](http://nodejs.org/documentation/)
* [API Docs](http://nodejs.org/api/)
* [Tutorials](http://nodejs.org/documentation/tutorials/)
* [Localization](http://nodejs.org/documentation/localization/)
* [Community](http://nodejs.org/community/)
* [Github Issues](https://github.com/joyent/node/issues)
* [Mailing List](http://groups.google.com/group/nodejs)
* [IRC](http://webchat.freenode.net/?channels=node.js)
* [Twitter](https://twitter.com/nodejs)
* [About](http://nodejs.org/about/)
* [Organization](http://nodejs.org/about/organization/)
* [Core Team](http://nodejs.org/about/core-team/)
* [Resources](http://nodejs.org/about/resources/)
* [Blog](http://blog.nodejs.org)

Copyright 2014 **[Joyent, Inc](http://joyent.com/)**, Node.js is a [**trademark**](https://nodejs.org/images/trademark-policy.pdf) of Joyent, Inc. [**View license**](https://raw.github.com/joyent/node/v0.10.35/LICENSE).