**ECMAScript Modules**[**#**](https://nodejs.org/api/esm.html#esm_ecmascript_modules)

[Stability: 1](https://nodejs.org/api/documentation.html#documentation_stability_index) - Experimental

**Introduction**[**#**](https://nodejs.org/api/esm.html#esm_introduction)

ECMAScript modules are [the official standard format](https://tc39.github.io/ecma262/#sec-modules) to package JavaScript code for reuse. Modules are defined using a variety of [import](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/import) and [export](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/export) statements.

Node.js fully supports ECMAScript modules as they are currently specified and provides limited interoperability between them and the existing module format, [CommonJS](https://nodejs.org/api/modules.html).

Node.js contains support for ES Modules based upon the [Node.js EP for ES Modules](https://github.com/nodejs/node-eps/blob/master/002-es-modules.md) and the [ECMAScript-modules implementation](https://github.com/nodejs/modules/blob/master/doc/plan-for-new-modules-implementation.md).

Expect major changes in the implementation including interoperability support, specifier resolution, and default behavior.

**Enabling**[**#**](https://nodejs.org/api/esm.html#esm_enabling)

The --experimental-modules flag can be used to enable support for ECMAScript modules (ES modules).

Once enabled, Node.js will treat the following as ES modules when passed to node as the initial input, or when referenced by import statements within ES module code:

* Files ending in .mjs.
* Files ending in .js, or extensionless files, when the nearest parent package.json file contains a top-level field "type" with a value of "module".
* Strings passed in as an argument to --eval or --print, or piped to node via STDIN, with the flag --input-type=module.

Node.js will treat as CommonJS all other forms of input, such as .js files where the nearest parent package.json file contains no top-level "type" field, or string input without the flag --input-type. This behavior is to preserve backward compatibility. However, now that Node.js supports both CommonJS and ES modules, it is best to be explicit whenever possible. Node.js will treat the following as CommonJS when passed to node as the initial input, or when referenced by import statements within ES module code:

* Files ending in .cjs.
* Files ending in .js, or extensionless files, when the nearest parent package.json file contains a top-level field "type" with a value of "commonjs".
* Strings passed in as an argument to --eval or --print, or piped to node via STDIN, with the flag --input-type=commonjs.

**package.json "type" field**[**#**](https://nodejs.org/api/esm.html#esm_code_package_json_code_code_type_code_field)

Files ending with .js or .mjs, or lacking any extension, will be loaded as ES modules when the nearest parent package.json file contains a top-level field "type" with a value of "module".

The nearest parent package.json is defined as the first package.json found when searching in the current folder, that folder’s parent, and so on up until the root of the volume is reached.

// package.json

{

"type": "module"

}

# In same folder as above package.json

node --experimental-modules my-app.js # Runs as ES module

If the nearest parent package.json lacks a "type" field, or contains "type": "commonjs", extensionless and .js files are treated as CommonJS. If the volume root is reached and no package.json is found, Node.js defers to the default, a package.json with no "type" field.

import statements of .js and extensionless files are treated as ES modules if the nearest parent package.json contains "type": "module".

// my-app.js, part of the same example as above

import './startup.js'; // Loaded as ES module because of package.json

Package authors should include the "type" field, even in packages where all sources are CommonJS. Being explicit about the type of the package will future-proof the package in case the default type of Node.js ever changes, and it will also make things easier for build tools and loaders to determine how the files in the package should be interpreted.

**Package Scope and File Extensions**[**#**](https://nodejs.org/api/esm.html#esm_package_scope_and_file_extensions)

A folder containing a package.json file, and all subfolders below that folder down until the next folder containing another package.json, is considered a *package scope*. The "type" field defines how .js and extensionless files should be treated within a particular package.json file’s package scope. Every package in a project’s node\_modules folder contains its own package.json file, so each project’s dependencies have their own package scopes. A package.json lacking a "type" field is treated as if it contained "type": "commonjs".

The package scope applies not only to initial entry points (node --experimental-modules my-app.js) but also to files referenced by import statements and import() expressions.

// my-app.js, in an ES module package scope because there is a package.json

// file in the same folder with "type": "module".

import './startup/init.js';

// Loaded as ES module since ./startup contains no package.json file,

// and therefore inherits the ES module package scope from one level up.

import 'commonjs-package';

// Loaded as CommonJS since ./node\_modules/commonjs-package/package.json

// lacks a "type" field or contains "type": "commonjs".

import './node\_modules/commonjs-package/index.js';

// Loaded as CommonJS since ./node\_modules/commonjs-package/package.json

// lacks a "type" field or contains "type": "commonjs".

Files ending with .mjs are always loaded as ES modules regardless of package scope.

Files ending with .cjs are always loaded as CommonJS regardless of package scope.

import './legacy-file.cjs';

// Loaded as CommonJS since .cjs is always loaded as CommonJS.

import 'commonjs-package/src/index.mjs';

// Loaded as ES module since .mjs is always loaded as ES module.

The .mjs and .cjs extensions may be used to mix types within the same package scope:

* Within a "type": "module" package scope, Node.js can be instructed to interpret a particular file as CommonJS by naming it with a .cjs extension (since both .js and .mjs files are treated as ES modules within a "module" package scope).
* Within a "type": "commonjs" package scope, Node.js can be instructed to interpret a particular file as an ES module by naming it with an .mjs extension (since both .js and .cjs files are treated as CommonJS within a "commonjs" package scope).

**--input-type flag**[**#**](https://nodejs.org/api/esm.html#esm_code_input_type_code_flag)

Strings passed in as an argument to --eval or --print (or -e or -p), or piped to node via STDIN, will be treated as ES modules when the --input-type=module flag is set.

node --experimental-modules --input-type=module --eval \

"import { sep } from 'path'; console.log(sep);"

echo "import { sep } from 'path'; console.log(sep);" | \

node --experimental-modules --input-type=module

For completeness there is also --input-type=commonjs, for explicitly running string input as CommonJS. This is the default behavior if --input-type is unspecified.

**Packages**[**#**](https://nodejs.org/api/esm.html#esm_packages)

**Package Entry Points**[**#**](https://nodejs.org/api/esm.html#esm_package_entry_points)

The package.json "main" field defines the entry point for a package, whether the package is included into CommonJS via require or into an ES module via import.

// ./node\_modules/es-module-package/package.json

{

"type": "module",

"main": "./src/index.js"

}

// ./my-app.mjs

import { something } from 'es-module-package';

// Loads from ./node\_modules/es-module-package/src/index.js

An attempt to require the above es-module-package would attempt to load ./node\_modules/es-module-package/src/index.js as CommonJS, which would throw an error as Node.js would not be able to parse the export statement in CommonJS.

As with import statements, for ES module usage the value of "main" must be a full path including extension: "./index.mjs", not "./index".

If the package.json "type" field is omitted, a .js file in "main" will be interpreted as CommonJS.

The "main" field can point to exactly one file, regardless of whether the package is referenced via require (in a CommonJS context) or import (in an ES module context).

**Compatibility with CommonJS-Only Versions of Node.js**[**#**](https://nodejs.org/api/esm.html#esm_compatibility_with_commonjs_only_versions_of_node_js)

Prior to the introduction of support for ES modules in Node.js, it was a common pattern for package authors to include both CommonJS and ES module JavaScript sources in their package, with package.json "main" specifying the CommonJS entry point and package.json "module" specifying the ES module entry point. This enabled Node.js to run the CommonJS entry point while build tools such as bundlers used the ES module entry point, since Node.js ignored (and still ignores) "module".

Node.js can now run ES module entry points, but it remains impossible for a package to define separate CommonJS and ES module entry points. This is for good reason: the pkg variable created from import pkg from 'pkg' is not the same singleton as the pkg variable created from const pkg = require('pkg'), so if both are referenced within the same app (including dependencies), unexpected behavior might occur.

There are two general approaches to addressing this limitation while still publishing a package that contains both CommonJS and ES module sources:

1. Document a new ES module entry point that’s not the package "main", e.g. import pkg from 'pkg/module.mjs' (or import 'pkg/esm', if using [package exports](https://nodejs.org/api/esm.html#esm_package_exports)). The package "main" would still point to a CommonJS file, and thus the package would remain compatible with older versions of Node.js that lack support for ES modules.
2. Switch the package "main" entry point to an ES module file as part of a breaking change version bump. This version and above would only be usable on ES module-supporting versions of Node.js. If the package still contains a CommonJS version, it would be accessible via a path within the package, e.g. require('pkg/commonjs'); this is essentially the inverse of the previous approach. Package consumers who are using CommonJS-only versions of Node.js would need to update their code from require('pkg') to e.g. require('pkg/commonjs').

Of course, a package could also include only CommonJS or only ES module sources. An existing package could make a semver major bump to an ES module-only version, that would only be supported in ES module-supporting versions of Node.js (and other runtimes). New packages could be published containing only ES module sources, and would be compatible only with ES module-supporting runtimes.

**Package Exports**[**#**](https://nodejs.org/api/esm.html#esm_package_exports)

By default, all subpaths from a package can be imported (import 'pkg/x.js'). Custom subpath aliasing and encapsulation can be provided through the "exports" field.

// ./node\_modules/es-module-package/package.json

{

"exports": {

"./submodule": "./src/submodule.js"

}

}

import submodule from 'es-module-package/submodule';

// Loads ./node\_modules/es-module-package/src/submodule.js

In addition to defining an alias, subpaths not defined by "exports" will throw when an attempt is made to import them:

import submodule from 'es-module-package/private-module.js';

// Throws ERR\_MODULE\_NOT\_FOUND

Note: this is not a strong encapsulation as any private modules can still be loaded by absolute paths.

Folders can also be mapped with package exports:

// ./node\_modules/es-module-package/package.json

{

"exports": {

"./features/": "./src/features/"

}

}

import feature from 'es-module-package/features/x.js';

// Loads ./node\_modules/es-module-package/src/features/x.js

If a package has no exports, setting "exports": false can be used instead of "exports": {} to indicate the package does not intend for submodules to be exposed.

Exports can also be used to map the main entry point of a package:

// ./node\_modules/es-module-package/package.json

{

"exports": {

".": "./main.js"

}

}

where the "." indicates loading the package without any subpath. Exports will always override any existing "main" value for both CommonJS and ES module packages.

For packages with only a main entry point, an "exports" value of just a string is also supported:

// ./node\_modules/es-module-package/package.json

{

"exports": "./main.js"

}

Any invalid exports entries will be ignored. This includes exports not starting with "./" or a missing trailing "/" for directory exports.

Array fallback support is provided for exports, similarly to import maps in order to be forward-compatible with fallback workflows in future:

{

"exports": {

"./submodule": ["not:valid", "./submodule.js"]

}

}

Since "not:valid" is not a supported target, "./submodule.js" is used instead as the fallback, as if it were the only target.

**import Specifiers**[**#**](https://nodejs.org/api/esm.html#esm_code_import_code_specifiers)

**Terminology**[**#**](https://nodejs.org/api/esm.html#esm_terminology)

The *specifier* of an import statement is the string after the from keyword, e.g. 'path' in import { sep } from 'path'. Specifiers are also used in export from statements, and as the argument to an import() expression.

There are four types of specifiers:

* *Bare specifiers* like 'some-package'. They refer to an entry point of a package by the package name.
* *Deep import specifiers* like 'some-package/lib/shuffle.mjs'. They refer to a path within a package prefixed by the package name.
* *Relative specifiers* like './startup.js' or '../config.mjs'. They refer to a path relative to the location of the importing file.
* *Absolute specifiers* like 'file:///opt/nodejs/config.js'. They refer directly and explicitly to a full path.

Bare specifiers, and the bare specifier portion of deep import specifiers, are strings; but everything else in a specifier is a URL.

Only file: and data: URLs are supported. A specifier like 'https://example.com/app.js' may be supported by browsers but it is not supported in Node.js.

Specifiers may not begin with / or //. These are reserved for potential future use. The root of the current volume may be referenced via file:///.

**data: Imports**[**#**](https://nodejs.org/api/esm.html#esm_data_imports)

Added in: v12.10.0

[data: URLs](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/Data_URIs) are supported for importing with the following MIME types:

* text/javascript for ES Modules
* application/json for JSON
* application/wasm for WASM.

data: URLs only resolve [*Bare specifiers*](https://nodejs.org/api/esm.html#esm_terminology) for builtin modules and [*Absolute specifiers*](https://nodejs.org/api/esm.html#esm_terminology). Resolving [*Relative specifiers*](https://nodejs.org/api/esm.html#esm_terminology) will not work because data: is not a [special scheme](https://url.spec.whatwg.org/#special-scheme). For example, attempting to load ./foo from data:text/javascript,import "./foo"; will fail to resolve since there is no concept of relative resolution for data: URLs. An example of a data: URLs being used is:

import 'data:text/javascript,console.log("hello!");';

import \_ from 'data:application/json,"world!"';

**import.meta**[**#**](https://nodejs.org/api/esm.html#esm_import_meta)

* [<Object>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object)

The import.meta metaproperty is an Object that contains the following property:

* url [<string>](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Data_structures#String_type) The absolute file: URL of the module.

**Differences Between ES Modules and CommonJS**[**#**](https://nodejs.org/api/esm.html#esm_differences_between_es_modules_and_commonjs)

**Mandatory file extensions**[**#**](https://nodejs.org/api/esm.html#esm_mandatory_file_extensions)

A file extension must be provided when using the import keyword. Directory indexes (e.g. './startup/index.js') must also be fully specified.

This behavior matches how import behaves in browser environments, assuming a typically configured server.

**No NODE\_PATH**[**#**](https://nodejs.org/api/esm.html#esm_no_code_node_path_code)

NODE\_PATH is not part of resolving import specifiers. Please use symlinks if this behavior is desired.

**No require, exports, module.exports, \_\_filename, \_\_dirname**[**#**](https://nodejs.org/api/esm.html#esm_no_code_require_code_code_exports_code_code_module_exports_code_code_filename_code_code_dirname_code)

These CommonJS variables are not available in ES modules.

require can be imported into an ES module using [module.createRequire()](https://nodejs.org/api/modules.html#modules_module_createrequire_filename).

Equivalents of \_\_filename and \_\_dirname can be created inside of each file via [import.meta.url](https://nodejs.org/api/esm.html#esm_import_meta).

import { fileURLToPath } from 'url';

import { dirname } from 'path';

const \_\_filename = fileURLToPath(import.meta.url);

const \_\_dirname = dirname(\_\_filename);

**No require.extensions**[**#**](https://nodejs.org/api/esm.html#esm_no_code_require_extensions_code)

require.extensions is not used by import. The expectation is that loader hooks can provide this workflow in the future.

**No require.cache**[**#**](https://nodejs.org/api/esm.html#esm_no_code_require_cache_code)

require.cache is not used by import. It has a separate cache.

**URL-based paths**[**#**](https://nodejs.org/api/esm.html#esm_url_based_paths)

ES modules are resolved and cached based upon [URL](https://url.spec.whatwg.org/) semantics. This means that files containing special characters such as # and ? need to be escaped.

Modules will be loaded multiple times if the import specifier used to resolve them have a different query or fragment.

import './foo.mjs?query=1'; // loads ./foo.mjs with query of "?query=1"

import './foo.mjs?query=2'; // loads ./foo.mjs with query of "?query=2"

For now, only modules using the file: protocol can be loaded.

**Interoperability with CommonJS**[**#**](https://nodejs.org/api/esm.html#esm_interoperability_with_commonjs)

**require**[**#**](https://nodejs.org/api/esm.html#esm_code_require_code)

require always treats the files it references as CommonJS. This applies whether require is used the traditional way within a CommonJS environment, or in an ES module environment using [module.createRequire()](https://nodejs.org/api/modules.html#modules_module_createrequire_filename).

To include an ES module into CommonJS, use [import()](https://nodejs.org/api/esm.html#esm_import-expressions).

**import statements**[**#**](https://nodejs.org/api/esm.html#esm_code_import_code_statements)

An import statement can reference an ES module, a CommonJS module, or JSON. Other file types such as Native modules are not supported. For those, use [module.createRequire()](https://nodejs.org/api/modules.html#modules_module_createrequire_filename).

import statements are permitted only in ES modules. For similar functionality in CommonJS, see [import()](https://nodejs.org/api/esm.html#esm_import-expressions).

The *specifier* of an import statement (the string after the from keyword) can either be an URL-style relative path like './file.mjs' or a package name like 'fs'.

Like in CommonJS, files within packages can be accessed by appending a path to the package name.

import { sin, cos } from 'geometry/trigonometry-functions.mjs';

Currently only the “default export” is supported for CommonJS files or packages:

import packageMain from 'commonjs-package'; // Works

import { method } from 'commonjs-package'; // Errors

There are ongoing efforts to make the latter code possible.

**import() expressions**[**#**](https://nodejs.org/api/esm.html#esm_code_import_code_expressions)

Dynamic import() is supported in both CommonJS and ES modules. It can be used to include ES module files from CommonJS code.

(async () => {

await import('./my-app.mjs');

})();

**CommonJS, JSON, and Native Modules**[**#**](https://nodejs.org/api/esm.html#esm_commonjs_json_and_native_modules)

CommonJS, JSON, and Native modules can be used with [module.createRequire()](https://nodejs.org/api/modules.html#modules_module_createrequire_filename).

// cjs.js

module.exports = 'cjs';

// esm.mjs

import { createRequire } from 'module';

import { fileURLToPath as fromURL } from 'url';

const require = createRequire(fromURL(import.meta.url));

const cjs = require('./cjs');

cjs === 'cjs'; // true

**Builtin modules**[**#**](https://nodejs.org/api/esm.html#esm_builtin_modules)

Builtin modules will provide named exports of their public API. A default export is also provided which is the value of the CommonJS exports. The default export can be used for, among other things, modifying the named exports. Named exports of builtin modules are updated only by calling [module.syncBuiltinESMExports()](https://nodejs.org/api/modules.html#modules_module_syncbuiltinesmexports).

import EventEmitter from 'events';

const e = new EventEmitter();

import { readFile } from 'fs';

readFile('./foo.txt', (err, source) => {

if (err) {

console.error(err);

} else {

console.log(source);

}

});

import fs, { readFileSync } from 'fs';

import { syncBuiltinESMExports } from 'module';

fs.readFileSync = () => Buffer.from('Hello, ESM');

syncBuiltinESMExports();

fs.readFileSync === readFileSync;

**Experimental JSON Modules**[**#**](https://nodejs.org/api/esm.html#esm_experimental_json_modules)

Currently importing JSON modules are only supported in the commonjs mode and are loaded using the CJS loader. [WHATWG JSON modules specification](https://html.spec.whatwg.org/#creating-a-json-module-script) are still being standardized, and are experimentally supported by including the additional flag --experimental-json-modules when running Node.js.

When the --experimental-json-modules flag is included both the commonjs and module mode will use the new experimental JSON loader. The imported JSON only exposes a default, there is no support for named exports. A cache entry is created in the CommonJS cache, to avoid duplication. The same object will be returned in CommonJS if the JSON module has already been imported from the same path.

Assuming an index.mjs with

import packageConfig from './package.json';

The --experimental-json-modules flag is needed for the module to work.

node --experimental-modules index.mjs # fails

node --experimental-modules --experimental-json-modules index.mjs # works

**Experimental Wasm Modules**[**#**](https://nodejs.org/api/esm.html#esm_experimental_wasm_modules)

Importing Web Assembly modules is supported under the --experimental-wasm-modules flag, allowing any .wasm files to be imported as normal modules while also supporting their module imports.

This integration is in line with the [ES Module Integration Proposal for Web Assembly](https://github.com/webassembly/esm-integration).

For example, an index.mjs containing:

import \* as M from './module.wasm';

console.log(M);

executed under:

node --experimental-modules --experimental-wasm-modules index.mjs

would provide the exports interface for the instantiation of module.wasm.

**Experimental Loader hooks**[**#**](https://nodejs.org/api/esm.html#esm_experimental_loader_hooks)

**Note: This API is currently being redesigned and will still change.**

To customize the default module resolution, loader hooks can optionally be provided via a --experimental-loader ./loader-name.mjs argument to Node.js.

When hooks are used they only apply to ES module loading and not to any CommonJS modules loaded.

**Resolve hook**[**#**](https://nodejs.org/api/esm.html#esm_resolve_hook)

The resolve hook returns the resolved file URL and module format for a given module specifier and parent file URL:

import { URL, pathToFileURL } from 'url';

const baseURL = pathToFileURL(process.cwd()).href;

/\*\*

\* @param {string} specifier

\* @param {string} parentModuleURL

\* @param {function} defaultResolver

\*/

export async function resolve(specifier,

parentModuleURL = baseURL,

defaultResolver) {

return {

url: new URL(specifier, parentModuleURL).href,

format: 'module'

};

}

The parentModuleURL is provided as undefined when performing main Node.js load itself.

The default Node.js ES module resolution function is provided as a third argument to the resolver for easy compatibility workflows.

In addition to returning the resolved file URL value, the resolve hook also returns a format property specifying the module format of the resolved module. This can be one of the following:

| **format** | **Description** |
| --- | --- |
| 'builtin' | Load a Node.js builtin module |
| 'commonjs' | Load a Node.js CommonJS module |
| 'dynamic' | Use a [dynamic instantiate hook](https://nodejs.org/api/esm.html#esm_dynamic_instantiate_hook) |
| 'json' | Load a JSON file |
| 'module' | Load a standard JavaScript module |
| 'wasm' | Load a WebAssembly module |

For example, a dummy loader to load JavaScript restricted to browser resolution rules with only JS file extension and Node.js builtin modules support could be written:

import path from 'path';

import process from 'process';

import Module from 'module';

import { URL, pathToFileURL } from 'url';

const builtins = Module.builtinModules;

const JS\_EXTENSIONS = new Set(['.js', '.mjs']);

const baseURL = pathToFileURL(process.cwd()).href;

/\*\*

\* @param {string} specifier

\* @param {string} parentModuleURL

\* @param {function} defaultResolver

\*/

export async function resolve(specifier,

parentModuleURL = baseURL,

defaultResolver) {

if (builtins.includes(specifier)) {

return {

url: specifier,

format: 'builtin'

};

}

if (/^\.{0,2}[/]/.test(specifier) !== true && !specifier.startsWith('file:')) {

// For node\_modules support:

// return defaultResolver(specifier, parentModuleURL);

throw new Error(

`imports must begin with '/', './', or '../'; '${specifier}' does not`);

}

const resolved = new URL(specifier, parentModuleURL);

const ext = path.extname(resolved.pathname);

if (!JS\_EXTENSIONS.has(ext)) {

throw new Error(

`Cannot load file with non-JavaScript file extension ${ext}.`);

}

return {

url: resolved.href,

format: 'module'

};

}

With this loader, running:

NODE\_OPTIONS='--experimental-modules --experimental-loader ./custom-loader.mjs' node x.js

would load the module x.js as an ES module with relative resolution support (with node\_modules loading skipped in this example).

**Dynamic instantiate hook**[**#**](https://nodejs.org/api/esm.html#esm_dynamic_instantiate_hook)

To create a custom dynamic module that doesn't correspond to one of the existing format interpretations, the dynamicInstantiate hook can be used. This hook is called only for modules that return format: 'dynamic' from the resolve hook.

export async function dynamicInstantiate(url) {

return {

exports: ['customExportName'],

execute: (exports) => {

// Get and set functions provided for pre-allocated export names

exports.customExportName.set('value');

}

};

}

With the list of module exports provided upfront, the execute function will then be called at the exact point of module evaluation order for that module in the import tree.

**Resolution Algorithm**[**#**](https://nodejs.org/api/esm.html#esm_resolution_algorithm)

**Features**[**#**](https://nodejs.org/api/esm.html#esm_features)

The resolver has the following properties:

* FileURL-based resolution as is used by ES modules
* Support for builtin module loading
* Relative and absolute URL resolution
* No default extensions
* No folder mains
* Bare specifier package resolution lookup through node\_modules

**Resolver Algorithm**[**#**](https://nodejs.org/api/esm.html#esm_resolver_algorithm)

The algorithm to load an ES module specifier is given through the **ESM\_RESOLVE** method below. It returns the resolved URL for a module specifier relative to a parentURL, in addition to the unique module format for that resolved URL given by the **ESM\_FORMAT** routine.

The *"module"* format is returned for an ECMAScript Module, while the *"commonjs"* format is used to indicate loading through the legacy CommonJS loader. Additional formats such as *"addon"* can be extended in future updates.

In the following algorithms, all subroutine errors are propagated as errors of these top-level routines unless stated otherwise.

*isMain* is **true** when resolving the Node.js application entry point.

Resolver algorithm specification

**Customizing ESM specifier resolution algorithm**[**#**](https://nodejs.org/api/esm.html#esm_customizing_esm_specifier_resolution_algorithm)

The current specifier resolution does not support all default behavior of the CommonJS loader. One of the behavior differences is automatic resolution of file extensions and the ability to import directories that have an index file.

The --es-module-specifier-resolution=[mode] flag can be used to customize the extension resolution algorithm. The default mode is explicit, which requires the full path to a module be provided to the loader. To enable the automatic extension resolution and importing from directories that include an index file use the node mode.

$ node --experimental-modules index.mjs

success!

$ node --experimental-modules index #Failure!

Error: Cannot find module

$ node --experimental-modules --es-module-specifier-resolution=node index

success!