Modules: ECMAScript modules

> *Stability: 2 - Stable*

Introduction

ECMAScript modules are the official standard format to package JavaScript code for reuse. Modules are defined using a variety of `import` and `export` statements.

The following example of an ES module exports a function:

```
// addTwo.mjs
function addTwo(num) {
  return num + 2;
}

export { addTwo };
```

The following example of an ES module imports the function from `addTwo.mjs`:

```
// app.mjs
import { addTwo } from './addTwo.mjs';

// Prints: 6
console.log(addTwo(4));
```

Node.js fully supports ECMAScript modules as they are currently specified and provides interoperability between them and its original module format, CommonJS.

Enabling

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

Authors can tell Node.js to use the ECMAScript modules loader via the `.mjs` file extension, the `package.json` `"type"` field, or the `--input-type` flag. Outside of those cases, Node.js will use the CommonJS module loader. See Determining module system for more details.

Packages

This section was moved to Modules: Packages.

`import` Specifiers

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 three types of specifiers:

- *Relative specifiers* like `'./startup.js'` or `'../config.mjs'`. They refer to a path relative to the location of the importing file. *The file extension is always necessary for these.*
- *Bare specifiers* like `'some-package'` or `'some-package/shuffle'`. They can refer to the main entry point of a package by the package name, or a specific feature module within a package prefixed by the package name as per the examples respectively. *Including the file extension is only necessary for packages without an `"exports"` field.*
- *Absolute specifiers* like `'file:///opt/nodejs/config.js'`. They refer directly and explicitly to a full path.

Bare specifier resolutions are handled by the Node.js module resolution algorithm. All other specifier resolutions are always only resolved with the standard relative URL resolution semantics.

Like in CommonJS, module files within packages can be accessed by appending a path to the package name unless the package's `package.json` contains an `"exports"` field, in which case files within packages can only be accessed via the paths defined in `"exports"`.

For details on these package resolution rules that apply to bare specifiers in the Node.js module resolution, see the packages documentation.

Mandatory file extensions

A file extension must be provided when using the `import` keyword to resolve relative or absolute specifiers. 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.

URLs

ES modules are resolved and cached as URLs. This means that special characters must be percent-encoded, such as `#` with `%23` and `?` with `%3F`.

`file:`, `node:`, and `data:` URL schemes are supported. A specifier like `'https://example.com/app.js'` is not supported natively in Node.js unless using a custom HTTPS loader.

`file:` URLs

Modules are loaded multiple times if the `import` specifier used to resolve them has 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"
```

The volume root may be referenced via `/`, `//` or `file:///`. Given the differences between URL and path resolution (such as percent encoding details), it is recommended to use url.pathToFileURL when importing a path.

`data:` Imports

`data:` URLs 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* for builtin modules and *Absolute specifiers*. Resolving *Relative specifiers* does not work because `data:` is not a special scheme. For example, attempting to load `./foo` from `data:text/javascript,import "./foo";` fails to resolve because 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!"';
```

`node:` Imports

`node:` URLs are supported as an alternative means to load Node.js builtin modules. This URL scheme allows for builtin modules to be referenced by valid absolute URL strings.

```
import fs from 'node:fs/promises';
```

Import assertions

> *Stability: 1 - Experimental*

The Import Assertions proposal adds an inline syntax for module import statements to pass on more information alongside the module specifier.

```
import fooData from './foo.json' assert { type: 'json' };

const { default: barData } =
  await import('./bar.json', { assert: { type: 'json' } });
```

Node.js supports the following `type` values, for which the assertion is mandatory:

Assertion `type`	Needed for
`'json'`	JSON modules

Builtin modules

Core modules 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()`.

```
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';
import { Buffer } from 'buffer';

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

fs.readFileSync === readFileSync;
```

`import()` expressions

Dynamic `import()` is supported in both CommonJS and ES modules. In CommonJS modules it can be used to load ES modules.

`import.meta`

- {Object}

The `import.meta` meta property is an `Object` that contains the following properties.

`import.meta.url`

- {string} The absolute `file:` URL of the module.

This is defined exactly the same as it is in browsers providing the URL of the current module file.

This enables useful patterns such as relative file loading:

```
import { readFileSync } from 'fs';
const buffer = readFileSync(new URL('./data.proto', import.meta.url));
```

`import.meta.resolve(specifier[, parent])`

> *Stability: 1 - Experimental*

This feature is only available with the `--experimental-import-meta-resolve` command flag enabled.

- `specifier` {string} The module specifier to resolve relative to `parent`.
- `parent` {string|URL} The absolute parent module URL to resolve from. If none is specified, the value of `import.meta.url` is used as the default.
- Returns: {Promise}

Provides a module-relative resolution function scoped to each module, returning the URL string.

```
const dependencyAsset = await import.meta.resolve('component-lib/asset.css');
```

`import.meta.resolve` also accepts a second argument which is the parent module from which to resolve from:

```
await import.meta.resolve('./dep', import.meta.url);
```

This function is asynchronous because the ES module resolver in Node.js is allowed to be asynchronous.

Interoperability with CommonJS

`import` statements

An `import` statement can reference an ES module or a CommonJS module. `import` statements are permitted only in ES modules, but dynamic `import()` expressions are supported in CommonJS for loading ES modules.

When importing CommonJS modules, the `module.exports` object is provided as the default export. Named exports may be available, provided by static analysis as a convenience for better ecosystem compatibility.

`require`

The CommonJS module `require` always treats the files it references as CommonJS.

Using `require` to load an ES module is not supported because ES modules have asynchronous execution. Instead, use `import()` to load an ES module from a CommonJS module.

CommonJS Namespaces

CommonJS modules consist of a `module.exports` object which can be of any type.

When importing a CommonJS module, it can be reliably imported using the ES module default import or its corresponding sugar syntax:

```
import { default as cjs } from 'cjs';

// The following import statement is "syntax sugar" (equivalent but sweeter)
// for `{ default as cjsSugar }` in the above import statement:
import cjsSugar from 'cjs';

console.log(cjs);
console.log(cjs === cjsSugar);
// Prints:
//   <module.exports>
//   true
```

The ECMAScript Module Namespace representation of a CommonJS module is always a namespace with a `default` export key pointing to the CommonJS `module.exports` value.

This Module Namespace Exotic Object can be directly observed either when using `import * as m from 'cjs'` or a dynamic import:

```
import * as m from 'cjs';
console.log(m);
console.log(m === await import('cjs'));
// Prints:
```

```
//   [Module] { default: <module.exports> }
//   true
```

For better compatibility with existing usage in the JS ecosystem, Node.js in addition attempts to determine the CommonJS named exports of every imported CommonJS module to provide them as separate ES module exports using a static analysis process.

For example, consider a CommonJS module written:

```
// cjs.cjs
exports.name = 'exported';
```

The preceding module supports named imports in ES modules:

```
import { name } from './cjs.cjs';
console.log(name);
// Prints: 'exported'

import cjs from './cjs.cjs';
console.log(cjs);
// Prints: { name: 'exported' }

import * as m from './cjs.cjs';
console.log(m);
// Prints: [Module] { default: { name: 'exported' }, name: 'exported' }
```

As can be seen from the last example of the Module Namespace Exotic Object being logged, the `name` export is copied off of the `module.exports` object and set directly on the ES module namespace when the module is imported.

Live binding updates or new exports added to `module.exports` are not detected for these named exports.

The detection of named exports is based on common syntax patterns but does not always correctly detect named exports. In these cases, using the default import form described above can be a better option.

Named exports detection covers many common export patterns, reexport patterns and build tool and transpiler outputs. See cjs-module-lexer for the exact semantics implemented.

Differences between ES modules and CommonJS

No `require`, `exports` or `module.exports`

In most cases, the ES module `import` can be used to load CommonJS modules.

If needed, a `require` function can be constructed within an ES module using `module.createRequire()`.

No `__filename` or `__dirname`

These CommonJS variables are not available in ES modules.

`__filename` and `__dirname` use cases can be replicated via `import.meta.url`.

No Native Module Loading

Native modules are not currently supported with ES module imports.

They can instead be loaded with `module.createRequire()` or `process.dlopen`.

No `require.resolve`

Relative resolution can be handled via `new URL('./local', import.meta.url)`.

For a complete `require.resolve` replacement, there is a flagged experimental `import.meta.resolve` API.

Alternatively `module.createRequire()` can be used.

No `NODE_PATH`

`NODE_PATH` is not part of resolving `import` specifiers. Please use symlinks if this behavior is desired.

No `require.extensions`

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

No `require.cache`

`require.cache` is not used by `import` as the ES module loader has its own separate cache.

JSON modules

> *Stability: 1 - Experimental*

JSON files can be referenced by `import`:

```
import packageConfig from './package.json' assert { type: 'json' };
```

The `assert { type: 'json' }` syntax is mandatory; see Import Assertions.

The imported JSON only exposes a `default` export. There is no support for named exports. A cache entry is created in the CommonJS cache to avoid duplication. The same object is returned in CommonJS if the JSON module has already been imported from the same path.

Wasm modules

> *Stability: 1 - Experimental*

Importing WebAssembly 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 WebAssembly.

For example, an `index.mjs` containing:

```
import * as M from './module.wasm';
console.log(M);
```

executed under:

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

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

Top-level `await`

> *Stability: 1 - Experimental*

The `await` keyword may be used in the top level body of an ECMAScript module.

Assuming an `a.mjs` with

```
export const five = await Promise.resolve(5);
```

And a `b.mjs` with

```
import { five } from './a.mjs';

console.log(five); // Logs `5`
```

```
node b.mjs # works
```

If a top level `await` expression never resolves, the `node` process will exit with a `13` status code.

```
import { spawn } from 'child_process';
import { execPath } from 'process';

spawn(execPath, [
  '--input-type=module',
  '--eval',
  // Never-resolving Promise:
  'await new Promise(() => {})',
]).once('exit', (code) => {
  console.log(code); // Logs `13`
});
```

HTTPS and HTTP imports

> *Stability: 1 - Experimental*

Importing network based modules using `https:` and `http:` is supported under the `--experimental-network-imports` flag. This allows web browser-like imports to work in Node.js with a few differences due to application stability and security concerns that are different when running in a privileged environment instead of a browser sandbox.

Imports are limited to HTTP/1

Automatic protocol negotiation for HTTP/2 and HTTP/3 is not yet supported.

HTTP is limited to loopback addresses

`http:` is vulnerable to man-in-the-middle attacks and is not allowed to be used for addresses outside of the IPv4 address `127.0.0.0/8` (`127.0.0.1` to `127.255.255.255`) and the IPv6 address `::1` . Support for `http:` is intended to be used for local development.

Authentication is never sent to the destination server.

`Authorization` , `Cookie` , and `Proxy-Authorization` headers are not sent to the server. Avoid including user info in parts of imported URLs. A security model for safely using these on the server is being worked on.

CORS is never checked on the destination server

CORS is designed to allow a server to limit the consumers of an API to a specific set of hosts. This is not supported as it does not make sense for a server-based implementation.

Cannot load non-network dependencies

These modules cannot access other modules that are not over `http:` or `https:` . To still access local modules while avoiding the security concern, pass in references to the local dependencies:

```
// file.mjs
import worker_threads from 'worker_threads';
import { configure, resize } from 'https://example.com/imagelib.mjs';
configure({ worker_threads });
```

```
// https://example.com/imagelib.mjs
let worker_threads;
export function configure(opts) {
  worker_threads = opts.worker_threads;
}
export function resize(img, size) {
  // Perform resizing in worker_thread to avoid main thread blocking
}
```

Network-based loading is not enabled by default

For now, the `--experimental-network-imports` flag is required to enable loading resources over `http:` or `https:` . In the future, a different mechanism will be used to enforce this. Opt-in is required to prevent transitive dependencies inadvertently using potentially mutable state that could affect reliability of Node.js applications.

Loaders

> *Stability: 1 - Experimental*

> *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 apply to the entry point and all `import` calls. They won't apply to `require` calls; those still follow CommonJS rules.

Hooks

`resolve(specifier, context, defaultResolve)`

> *The loaders API is being redesigned. This hook may disappear or its signature may change. Do not rely on the API described below.*

- `specifier` {string}
- `context` {Object}
  - `conditions` {string[]}
  - `importAssertions` {Object}
  - `parentURL` {string|undefined}
- `defaultResolve` {Function} The Node.js default resolver.
- Returns: {Object}
  - `format` {string|null|undefined} `'builtin' | 'commonjs' | 'json' | 'module' | 'wasm'`
  - `url` {string} The absolute url to the import target (such as `file://…`)

The `resolve` hook returns the resolved file URL for a given module specifier and parent URL, and optionally its format (such as `'module'`) as a hint to the `load` hook. If a format is specified, the `load` hook is ultimately responsible for providing the final `format` value (and it is free to ignore the hint provided by `resolve`); if `resolve` provides a `format`, a custom `load` hook is required even if only to pass the value to the Node.js default `load` hook.

The module specifier is the string in an `import` statement or `import()` expression, and the parent URL is the URL of the module that imported this one, or `undefined` if this is the main entry point for the application.

The `conditions` property in `context` is an array of conditions for package exports conditions that apply to this resolution request. They can be used for looking up conditional mappings elsewhere or to modify the list when calling the default resolution logic.

The current package exports conditions are always in the `context.conditions` array passed into the hook. To guarantee *default Node.js module specifier resolution behavior* when calling `defaultResolve`, the `context.conditions` array passed to it *must* include *all* elements of the `context.conditions` array originally passed into the `resolve` hook.

```
/**
 * @param {string} specifier
 * @param {{
 *   conditions: string[],
 *   parentURL: string | undefined,
 * }} context
 * @param {Function} defaultResolve
 * @returns {Promise<{ url: string }>}
 */
export async function resolve(specifier, context, defaultResolve) {
  const { parentURL = null } = context;
  if (Math.random() > 0.5) { // Some condition.
```

```
    // For some or all specifiers, do some custom logic for resolving.
    // Always return an object of the form {url: <string>}.
    return {
      url: parentURL ?
        new URL(specifier, parentURL).href :
        new URL(specifier).href,
    };
  }
  if (Math.random() < 0.5) { // Another condition.
    // When calling `defaultResolve`, the arguments can be modified. In this
    // case it's adding another value for matching conditional exports.
    return defaultResolve(specifier, {
      ...context,
      conditions: [...context.conditions, 'another-condition'],
    });
  }
  // Defer to Node.js for all other specifiers.
  return defaultResolve(specifier, context, defaultResolve);
}
```

`load(url, context, defaultLoad)`

> *The loaders API is being redesigned. This hook may disappear or its signature may change. Do not rely on the API described below.*

> *In a previous version of this API, this was split across 3 separate, now deprecated, hooks (`getFormat` , `getSource` , and `transformSource`).*

- `url` {string}
- `context` {Object}
  - `format` {string|null|undefined} The format optionally supplied by the `resolve` hook.
  - `importAssertions` {Object}
- `defaultLoad` {Function}
- Returns: {Object}
  - `format` {string}
  - `source` {string|ArrayBuffer|TypedArray}

The `load` hook provides a way to define a custom method of determining how a URL should be interpreted, retrieved, and parsed. It is also in charge of validating the import assertion.

The final value of `format` must be one of the following:

`format`	Description	Acceptable types for `source` returned by `load`
`'builtin'`	Load a Node.js builtin module	Not applicable
`'commonjs'`	Load a Node.js CommonJS module	Not applicable
`'json'`	Load a JSON file	{ `string`, `ArrayBuffer`, `TypedArray` }
`'module'`	Load an ES module	{ `string`, `ArrayBuffer`, `TypedArray` }
`'wasm'`	Load a WebAssembly module	{ `ArrayBuffer`, `TypedArray` }

The value of `source` is ignored for type `'builtin'` because currently it is not possible to replace the value of a Node.js builtin (core) module. The value of `source` is ignored for type `'commonjs'` because the CommonJS module loader does not provide a mechanism for the ES module loader to override the CommonJS module return value. This limitation might be overcome in the future.

> ***Caveat**: The ESM `load` hook and namespaced exports from CommonJS modules are incompatible. Attempting to use them together will result in an empty object from the import. This may be addressed in the future.*

> *These types all correspond to classes defined in ECMAScript.*

- The specific `ArrayBuffer` object is a `SharedArrayBuffer`.
- The specific `TypedArray` object is a `Uint8Array`.

If the source value of a text-based format (i.e., `'json'`, `'module'`) is not a string, it is converted to a string using `util.TextDecoder`.

The `load` hook provides a way to define a custom method for retrieving the source code of an ES module specifier. This would allow a loader to potentially avoid reading files from disk. It could also be used to map an unrecognized format to a supported one, for example `yaml` to `module`.

```
/**
 * @param {string} url
 * @param {{
    format: string,
  }} context If resolve settled with a `format`, that value is included here.
 * @param {Function} defaultLoad
 * @returns {Promise<{
    format: string,
    source: string | ArrayBuffer | SharedArrayBuffer | Uint8Array,
  }>}
 */
export async function load(url, context, defaultLoad) {
  const { format } = context;
  if (Math.random() > 0.5) { // Some condition.
    /*
      For some or all URLs, do some custom logic for retrieving the source.
      Always return an object of the form {
        format: <string>,
        source: <string|buffer>,
      }.
    */
    return {
      format,
      source: '...',
    };
  }
  // Defer to Node.js for all other URLs.
  return defaultLoad(url, context, defaultLoad);
}
```

In a more advanced scenario, this can also be used to transform an unsupported source to a supported one (see Examples below).

`globalPreload()`

> *The loaders API is being redesigned. This hook may disappear or its signature may change. Do not rely on the API described below.*

> *In a previous version of this API, this hook was named `getGlobalPreloadCode`.*

- Returns: {string}

Sometimes it might be necessary to run some code inside of the same global scope that the application runs in. This hook allows the return of a string that is run as a sloppy-mode script on startup.

Similar to how CommonJS wrappers work, the code runs in an implicit function scope. The only argument is a `require`-like function that can be used to load builtins like "fs": `getBuiltin(request: string)`.

If the code needs more advanced `require` features, it has to construct its own `require` using `module.createRequire()`.

```
/**
 * @param {{
     port: MessagePort,
   }} utilities Things that preload code might find useful
 * @returns {string} Code to run before application startup
 */
export function globalPreload(utilities) {
  return `\
globalThis.someInjectedProperty = 42;
console.log('I just set some globals!');

const { createRequire } = getBuiltin('module');
const { cwd } = getBuiltin('process');

const require = createRequire(cwd() + '/<preload>');
// [...]
`;
}
```

In order to allow communication between the application and the loader, another argument is provided to the preload code: `port`. This is available as a parameter to the loader hook and inside of the source text returned by the hook. Some care must be taken in order to properly call `port.ref()` and `port.unref()` to prevent a process from being in a state where it won't close normally.

```
/**
 * This example has the application context send a message to the loader
 * and sends the message back to the application context
 * @param {{
     port: MessagePort,
   }} utilities Things that preload code might find useful
 * @returns {string} Code to run before application startup
```

```
 */
export function globalPreload({ port }) {
  port.onmessage = (evt) => {
    port.postMessage(evt.data);
  };
  return `\
    port.postMessage('console.log("I went to the Loader and back");');
    port.onmessage = (evt) => {
      eval(evt.data);
    };
  `;
}
```

Examples

The various loader hooks can be used together to accomplish wide-ranging customizations of Node.js' code loading and evaluation behaviors.

HTTPS loader

In current Node.js, specifiers starting with `https://` are unsupported. The loader below registers hooks to enable rudimentary support for such specifiers. While this may seem like a significant improvement to Node.js core functionality, there are substantial downsides to actually using this loader: performance is much slower than loading files from disk, there is no caching, and there is no security.

```
// https-loader.mjs
import { get } from 'https';

export function resolve(specifier, context, defaultResolve) {
  const { parentURL = null } = context;

  // Normally Node.js would error on specifiers starting with 'https://', so
  // this hook intercepts them and converts them into absolute URLs to be
  // passed along to the later hooks below.
  if (specifier.startsWith('https://')) {
    return {
      url: specifier
    };
  } else if (parentURL && parentURL.startsWith('https://')) {
    return {
      url: new URL(specifier, parentURL).href
    };
  }

  // Let Node.js handle all other specifiers.
  return defaultResolve(specifier, context, defaultResolve);
}

export function load(url, context, defaultLoad) {
  // For JavaScript to be loaded over the network, we need to fetch and
  // return it.
  if (url.startsWith('https://')) {
```

```
    return new Promise((resolve, reject) => {
      get(url, (res) => {
        let data = '';
        res.on('data', (chunk) => data += chunk);
        res.on('end', () => resolve({
          // This example assumes all network-provided JavaScript is ES module
          // code.
          format: 'module',
          source: data,
        }));
      }).on('error', (err) => reject(err));
    });
  }

  // Let Node.js handle all other URLs.
  return defaultLoad(url, context, defaultLoad);
}
```

```
// main.mjs
import { VERSION } from 'https://coffeescript.org/browser-compiler-
modern/coffeescript.js';

console.log(VERSION);
```

With the preceding loader, running `node --experimental-loader ./https-loader.mjs ./main.mjs` prints the current version of CoffeeScript per the module at the URL in `main.mjs`.

Transpiler loader

Sources that are in formats Node.js doesn't understand can be converted into JavaScript using the `load` hook. Before that hook gets called, however, a `resolve` hook needs to tell Node.js not to throw an error on unknown file types.

This is less performant than transpiling source files before running Node.js; a transpiler loader should only be used for development and testing purposes.

```
// coffeescript-loader.mjs
import { readFile } from 'node:fs/promises';
import { dirname, extname, resolve as resolvePath } from 'node:path';
import { cwd } from 'node:process';
import { fileURLToPath, pathToFileURL } from 'node:url';
import CoffeeScript from 'coffeescript';

const baseURL = pathToFileURL(`${cwd()}/`).href;

// CoffeeScript files end in .coffee, .litcoffee or .coffee.md.
const extensionsRegex = /\.coffee$|\.litcoffee$|\.coffee\.md$/;

export async function resolve(specifier, context, defaultResolve) {
  const { parentURL = baseURL } = context;
```

```
  // Node.js normally errors on unknown file extensions, so return a URL for
  // specifiers ending in the CoffeeScript file extensions.
  if (extensionsRegex.test(specifier)) {
    return {
      url: new URL(specifier, parentURL).href
    };
  }

  // Let Node.js handle all other specifiers.
  return defaultResolve(specifier, context, defaultResolve);
}

export async function load(url, context, defaultLoad) {
  // Now that we patched resolve to let CoffeeScript URLs through, we need to
  // tell Node.js what format such URLs should be interpreted as. Because
  // CoffeeScript transpiles into JavaScript, it should be one of the two
  // JavaScript formats: 'commonjs' or 'module'.
  if (extensionsRegex.test(url)) {
    // CoffeeScript files can be either CommonJS or ES modules, so we want any
    // CoffeeScript file to be treated by Node.js the same as a .js file at the
    // same location. To determine how Node.js would interpret an arbitrary .js
    // file, search up the file system for the nearest parent package.json file
    // and read its "type" field.
    const format = await getPackageType(url);
    // When a hook returns a format of 'commonjs', `source` is be ignored.
    // To handle CommonJS files, a handler needs to be registered with
    // `require.extensions` in order to process the files with the CommonJS
    // loader. Avoiding the need for a separate CommonJS handler is a future
    // enhancement planned for ES module loaders.
    if (format === 'commonjs') {
      return { format };
    }

    const { source: rawSource } = await defaultLoad(url, { format });
    // This hook converts CoffeeScript source code into JavaScript source code
    // for all imported CoffeeScript files.
    const transformedSource = CoffeeScript.compile(rawSource.toString(), {
      bare: true,
      filename: url,
    });

    return {
      format,
      source: transformedSource,
    };
  }

  // Let Node.js handle all other URLs.
  return defaultLoad(url, context, defaultLoad);
}

async function getPackageType(url) {
```

```
  // `url` is only a file path during the first iteration when passed the
  // resolved url from the load() hook
  // an actual file path from load() will contain a file extension as it's
  // required by the spec
  // this simple truthy check for whether `url` contains a file extension will
  // work for most projects but does not cover some edge-cases (such as
  // extensionless files or a url ending in a trailing space)
  const isFilePath = !!extname(url);
  // If it is a file path, get the directory it's in
  const dir = isFilePath ?
    dirname(fileURLToPath(url)) :
    url;
  // Compose a file path to a package.json in the same directory,
  // which may or may not exist
  const packagePath = resolvePath(dir, 'package.json');
  // Try to read the possibly nonexistent package.json
  const type = await readFile(packagePath, { encoding: 'utf8' })
    .then((filestring) => JSON.parse(filestring).type)
    .catch((err) => {
      if (err?.code !== 'ENOENT') console.error(err);
    });
  // Ff package.json existed and contained a `type` field with a value, voila
  if (type) return type;
  // Otherwise, (if not at the root) continue checking the next directory up
  // If at the root, stop and return false
  return dir.length > 1 && getPackageType(resolvePath(dir, '..'));
}
```

```
# main.coffee
import { scream } from './scream.coffee'
console.log scream 'hello, world'

import { version } from 'process'
console.log "Brought to you by Node.js version #{version}"
```

```
# scream.coffee
export scream = (str) -> str.toUpperCase()
```

With the preceding loader, running `node --experimental-loader ./coffeescript-loader.mjs main.coffee` causes `main.coffee` to be turned into JavaScript after its source code is loaded from disk but before Node.js executes it; and so on for any `.coffee`, `.litcoffee` or `.coffee.md` files referenced via `import` statements of any loaded file.

Resolution algorithm

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

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.

The algorithm to determine the module format of a resolved URL is provided by **ESM_FORMAT**, which returns the unique module format for any file. 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.

defaultConditions is the conditional environment name array, `["node", "import"]`.

The resolver can throw the following errors:

- *Invalid Module Specifier*: Module specifier is an invalid URL, package name or package subpath specifier.
- *Invalid Package Configuration*: package.json configuration is invalid or contains an invalid configuration.
- *Invalid Package Target*: Package exports or imports define a target module for the package that is an invalid type or string target.
- *Package Path Not Exported*: Package exports do not define or permit a target subpath in the package for the given module.
- *Package Import Not Defined*: Package imports do not define the specifier.
- *Module Not Found*: The package or module requested does not exist.
- *Unsupported Directory Import*: The resolved path corresponds to a directory, which is not a supported target for module imports.

Resolver Algorithm Specification

ESM_RESOLVE(*specifier*, *parentURL*)

> *1. Let resolved be **undefined**.*
> *2. If specifier is a valid URL, then*
> *1. Set resolved to the result of parsing and reserializing specifier as a URL.*
> *3. Otherwise, if specifier starts with "/", "./" or "../", then*
> *1. Set resolved to the URL resolution of specifier relative to parentURL.*
> *4. Otherwise, if specifier starts with "#", then*
> *1. Set resolved to the result of **PACKAGE_IMPORTS_RESOLVE**(specifier, parentURL, defaultConditions).*
> *5. Otherwise,*
> *1. Note: specifier is now a bare specifier.*
> *2. Set resolved the result of **PACKAGE_RESOLVE**(specifier, parentURL).*
> *6. Let format be **undefined**.*
> *7. If resolved is a "file:" URL, then*
> *1. If resolved contains any percent encodings of "/" or "\" ("%2F" and "%5C" respectively), then*
> *1. Throw an Invalid Module Specifier error.*

2. *If the file at resolved is a directory, then*
    1. *Throw an Unsupported Directory Import error.*
3. *If the file at resolved does not exist, then*
    1. *Throw a Module Not Found error.*
4. *Set resolved to the real path of resolved, maintaining the same URL querystring and fragment components.*
5. *Set format to the result of **ESM_FILE_FORMAT**(resolved).*

8. *Otherwise,*
    1. *Set format the module format of the content type associated with the URL resolved.*
9. *Load resolved as module format, format.*

PACKAGE_RESOLVE(*packageSpecifier*, *parentURL*)

1. *Let packageName be **undefined**.*
2. *If packageSpecifier is an empty string, then*
    1. *Throw an Invalid Module Specifier error.*
3. *If packageSpecifier is a Node.js builtin module name, then*
    1. *Return the string "node:" concatenated with packageSpecifier.*
4. *If packageSpecifier does not start with "@", then*
    1. *Set packageName to the substring of packageSpecifier until the first "/" separator or the end of the string.*
5. *Otherwise,*
    1. *If packageSpecifier does not contain a "/" separator, then*
        1. *Throw an Invalid Module Specifier error.*
    2. *Set packageName to the substring of packageSpecifier until the second "/" separator or the end of the string.*
6. *If packageName starts with "." or contains "\" or "%", then*
    1. *Throw an Invalid Module Specifier error.*
7. *Let packageSubpath be "." concatenated with the substring of packageSpecifier from the position at the length of packageName.*
8. *If packageSubpath ends in "/", then*
    1. *Throw an Invalid Module Specifier error.*
9. *Let selfUrl be the result of **PACKAGE_SELF_RESOLVE**(packageName, packageSubpath, parentURL).*
10. *If selfUrl is not **undefined**, return selfUrl.*
11. *While parentURL is not the file system root,*
    1. *Let packageURL be the URL resolution of "node_modules/" concatenated with packageSpecifier, relative to parentURL.*
    2. *Set parentURL to the parent folder URL of parentURL.*
    3. *If the folder at packageURL does not exist, then*
        1. *Continue the next loop iteration.*
    4. *Let pjson be the result of **READ_PACKAGE_JSON**(packageURL).*
    5. *If pjson is not **null** and pjson.exports is not **null** or **undefined**, then*
        1. *Return the result of **PACKAGE_EXPORTS_RESOLVE**(packageURL, packageSubpath, pjson.exports, defaultConditions).*
    6. *Otherwise, if packageSubpath is equal to ".", then*
        1. *If pjson.main is a string, then*
            1. *Return the URL resolution of main in packageURL.*

7. *Otherwise,*
    1. *Return the URL resolution of packageSubpath in packageURL.*

12. *Throw a Module Not Found error.*

PACKAGE_SELF_RESOLVE(*packageName*, *packageSubpath*, *parentURL*)

1. *Let packageURL be the result of **LOOKUP_PACKAGE_SCOPE**(parentURL).*
2. *If packageURL is **null**, then*
    1. *Return **undefined**.*
3. *Let pjson be the result of **READ_PACKAGE_JSON**(packageURL).*
4. *If pjson is **null** or if pjson.exports is **null** or **undefined**, then*
    1. *Return **undefined**.*
5. *If pjson.name is equal to packageName, then*
    1. *Return the result of **PACKAGE_EXPORTS_RESOLVE**(packageURL, packageSubpath, pjson.exports, defaultConditions).*
6. *Otherwise, return **undefined**.*

PACKAGE_EXPORTS_RESOLVE(*packageURL*, *subpath*, *exports*, *conditions*)

1. *If exports is an Object with both a key starting with "." and a key not starting with ".", throw an Invalid Package Configuration error.*
2. *If subpath is equal to ".", then*
    1. *Let mainExport be **undefined**.*
    2. *If exports is a String or Array, or an Object containing no keys starting with ".", then*
        1. *Set mainExport to exports.*
    3. *Otherwise if exports is an Object containing a "." property, then*
        1. *Set mainExport to exports_["."_].*
    4. *If mainExport is not **undefined**, then*
        1. *Let resolved be the result of **PACKAGE_TARGET_RESOLVE**(packageURL, mainExport, "", **false**, **false**, conditions).*
        2. *If resolved is not **null** or **undefined**, return resolved.*
3. *Otherwise, if exports is an Object and all keys of exports start with ".", then*
    1. *Let matchKey be the string "./" concatenated with subpath.*
    2. *Let resolved be the result of **PACKAGE_IMPORTS_EXPORTS_RESOLVE**(matchKey, exports, packageURL, **false**, conditions).*
    3. *If resolved is not **null** or **undefined**, return resolved.*
4. *Throw a Package Path Not Exported error.*

PACKAGE_IMPORTS_RESOLVE(*specifier*, *parentURL*, *conditions*)

1. *Assert: specifier begins with "#".*
2. *If specifier is exactly equal to "#" or starts with "#/", then*
    1. *Throw an Invalid Module Specifier error.*
3. *Let packageURL be the result of **LOOKUP_PACKAGE_SCOPE**(parentURL).*
4. *If packageURL is not **null**, then*
    1. *Let pjson be the result of **READ_PACKAGE_JSON**(packageURL).*
    2. *If pjson.imports is a non-null Object, then*

1. Let resolved be the result of ***PACKAGE_IMPORTS_EXPORTS_RESOLVE****(specifier, pjson.imports, packageURL,* ***true****, conditions).*
2. *If resolved is not* ***null*** *or* ***undefined****, return resolved.*

5. *Throw a Package Import Not Defined error.*

PACKAGE_IMPORTS_EXPORTS_RESOLVE(*matchKey*, *matchObj*, *packageURL*, *isImports*, *conditions*)

1. *If matchKey is a key of matchObj and does not contain "*", then*
    1. *Let target be the value of _matchObj_[matchKey].*
    2. *Return the result of* ***PACKAGE_TARGET_RESOLVE****(packageURL, target, "",* ***false****, isImports, conditions).*
2. *Let expansionKeys be the list of keys of matchObj containing only a single "*", sorted by the sorting function* ***PATTERN_KEY_COMPARE*** *which orders in descending order of specificity.*
3. *For each key expansionKey in expansionKeys, do*
    1. *Let patternBase be the substring of expansionKey up to but excluding the first "*" character.*
    2. *If matchKey starts with but is not equal to patternBase, then*
        1. *Let patternTrailer be the substring of expansionKey from the index after the first "*" character.*
        2. *If patternTrailer has zero length, or if matchKey ends with patternTrailer and the length of matchKey is greater than or equal to the length of expansionKey, then*
            1. *Let target be the value of _matchObj_[expansionKey].*
            2. *Let subpath be the substring of matchKey starting at the index of the length of patternBase up to the length of matchKey minus the length of patternTrailer.*
            3. *Return the result of* ***PACKAGE_TARGET_RESOLVE****(packageURL, target, subpath,* ***true****, isImports, conditions).*

4. *Return* ***null****.*

PATTERN_KEY_COMPARE(*keyA*, *keyB*)

1. *Assert: keyA ends with "/" or contains only a single "*".*
2. *Assert: keyB ends with "/" or contains only a single "*".*
3. *Let baseLengthA be the index of "*" in keyA plus one, if keyA contains "*", or the length of keyA otherwise.*
4. *Let baseLengthB be the index of "*" in keyB plus one, if keyB contains "*", or the length of keyB otherwise.*
5. *If baseLengthA is greater than baseLengthB, return -1.*
6. *If baseLengthB is greater than baseLengthA, return 1.*
7. *If keyA does not contain "*", return 1.*
8. *If keyB does not contain "*", return -1.*
9. *If the length of keyA is greater than the length of keyB, return -1.*
10. *If the length of keyB is greater than the length of keyA, return 1.*
11. *Return 0.*

PACKAGE_TARGET_RESOLVE(*packageURL*, *target*, *subpath*, *pattern*, *internal*, *conditions*)

1. *If target is a String, then*
    1. *If pattern is* ***false****, subpath has non-zero length and target does not end with "/", throw an Invalid Module Specifier error.*
    2. *If target does not start with "./", then*
        1. *If internal is* ***true*** *and target does not start with "../" or "/" and is not a valid URL, then*
            1. *If pattern is* ***true****, then*

1. Return ***PACKAGE_RESOLVE***(target with every instance of "*" replaced by subpath, packageURL + "/").

2. Return ***PACKAGE_RESOLVE***(target + subpath, packageURL + "/").

2. Otherwise, throw an Invalid Package Target error.

3. If target split on "/" or "\" contains any ".", ".." or "node_modules" segments after the first segment, case insensitive and including percent encoded variants, throw an Invalid Package Target error.
4. Let resolvedTarget be the URL resolution of the concatenation of packageURL and target.
5. Assert: resolvedTarget is contained in packageURL.
6. If subpath split on "/" or "\" contains any ".", ".." or "node_modules" segments, case insensitive and including percent encoded variants, throw an Invalid Module Specifier error.
7. If pattern is ***true***, then
    1. Return the URL resolution of resolvedTarget with every instance of "*" replaced with subpath.
8. Otherwise,
    1. Return the URL resolution of the concatenation of subpath and resolvedTarget.

2. Otherwise, if target is a non-null Object, then
    1. If exports contains any index property keys, as defined in ECMA-262 [6.1.7 Array Index], throw an Invalid Package Configuration error.
    2. For each property p of target, in object insertion order as,
        1. If p equals "default" or conditions contains an entry for p, then
            1. Let targetValue be the value of the p property in target.
            2. Let resolved be the result of ***PACKAGE_TARGET_RESOLVE***(packageURL, targetValue, subpath, pattern, internal, conditions).
            3. If resolved is equal to ***undefined***, continue the loop.
            4. Return resolved.
    3. Return ***undefined***.
3. Otherwise, if target is an Array, then
    1. If _target.length is zero, return ***null***.
    2. For each item targetValue in target, do
        1. Let resolved be the result of ***PACKAGE_TARGET_RESOLVE***(packageURL, targetValue, subpath, pattern, internal, conditions), continuing the loop on any Invalid Package Target error.
        2. If resolved is ***undefined***, continue the loop.
        3. Return resolved.
    3. Return or throw the last fallback resolution ***null*** return or error.
4. Otherwise, if target is null, return ***null***.
5. Otherwise throw an Invalid Package Target error.

ESM_FILE_FORMAT(*url*)

1. Assert: url corresponds to an existing file.
2. If url ends in ".mjs", then
    1. Return "module".
3. If url ends in ".cjs", then
    1. Return "commonjs".
4. If url ends in ".json", then

> 1. *Return "json".*
> 5. *Let packageURL be the result of **LOOKUP_PACKAGE_SCOPE**(url).*
> 6. *Let pjson be the result of **READ_PACKAGE_JSON**(packageURL).*
> 7. *If pjson?.type exists and is "module", then*
>   1. *If url ends in ".js", then*
>   1. *Return "module".*
>   2. *Throw an Unsupported File Extension error.*
> 8. *Otherwise,*
>   1. *Throw an Unsupported File Extension error.*

LOOKUP_PACKAGE_SCOPE(*url*)

> 1. *Let scopeURL be url.*
> 2. *While scopeURL is not the file system root,*
>   1. *Set scopeURL to the parent URL of scopeURL.*
>   2. *If scopeURL ends in a "node_modules" path segment, return **null**.*
>   3. *Let pjsonURL be the resolution of "package.json" within scopeURL.*
>   4. *if the file at pjsonURL exists, then*
>   1. *Return scopeURL.*
> 3. *Return **null**.*

READ_PACKAGE_JSON(*packageURL*)

> 1. *Let pjsonURL be the resolution of "package.json" within packageURL.*
> 2. *If the file at pjsonURL does not exist, then*
>   1. *Return **null**.*
> 3. *If the file at packageURL does not parse as valid JSON, then*
>   1. *Throw an Invalid Package Configuration error.*
> 4. *Return the parsed JSON source of the file at pjsonURL.*

Customizing ESM specifier resolution algorithm

> *Stability: 1 - Experimental*

> *Do not rely on this flag. We plan to remove it once the Loaders API has advanced to the point that equivalent functionality can be achieved via custom loaders.*

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 `--experimental-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 index.mjs
success!
$ node index # Failure!
Error: Cannot find module
```

```
$ node --experimental-specifier-resolution=node index
success!
```