Modules: Packages

Introduction

A package is a folder tree described by a `package.json` file. The package consists of the folder containing the `package.json` file and all subfolders until the next folder containing another `package.json` file, or a folder named `node_modules`.

This page provides guidance for package authors writing `package.json` files along with a reference for the `package.json` fields defined by Node.js.

Determining module system

Node.js will treat the following as ES modules when passed to `node` as the initial input, or when referenced by `import` statements or `import()` expressions:

- Files with an `.mjs` extension.
- Files with a `.js` extension when the nearest parent `package.json` file contains a top-level `"type"` field with a value of `"module"`.
- Strings passed in as an argument to `--eval`, 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, `import()` expressions, or `require()` expressions:

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

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.

Modules loaders

Node.js has two systems for resolving a specifier and loading modules.

There is the CommonJS module loader:

- It is fully synchronous.

- It is responsible for handling `require()` calls.
- It is monkey patchable.
- It supports folders as modules.
- When resolving a specifier, if no exact match is found, it will try to add extensions (`.js` , `.json` , and finally `.node`) and then attempt to resolve folders as modules.
- It treats `.json` as JSON text files.
- `.node` files are interpreted as compiled addon modules loaded with `process.dlopen()` .
- It treats all files that lack `.json` or `.node` extensions as JavaScript text files.
- It cannot be used to load ECMAScript modules (although it is possible to load ECMAScript modules from CommonJS modules). When used to load a JavaScript text file that is not an ECMAScript module, it loads it as a CommonJS module.

There is the ECMAScript module loader:

- It is asynchronous.
- It is responsible for handling `import` statements and `import()` expressions.
- It is not monkey patchable, can be customized using loader hooks.
- It does not support folders as modules, directory indexes (e.g. `'./startup/index.js'`) must be fully specified.
- It does no extension searching. A file extension must be provided when the specifier is a relative or absolute file URL.
- It can load JSON modules, but an import assertion is required.
- It accepts only `.js` , `.mjs` , and `.cjs` extensions for JavaScript text files.
- It can be used to load JavaScript CommonJS modules. Such modules are passed through the `es-module-lexer` to try to identify named exports, which are available if they can be determined through static analysis. Imported CommonJS modules have their URLs converted to absolute paths and are then loaded via the CommonJS module loader.

`package.json` and file extensions

Within a package, the `package.json` `"type"` field defines how Node.js should interpret `.js` files. If a `package.json` file does not have a `"type"` field, `.js` files are treated as CommonJS.

A `package.json` `"type"` value of `"module"` tells Node.js to interpret `.js` files within that package as using ES module syntax.

The `"type"` field applies not only to initial entry points (`node my-app.js`) but also to files referenced by `import` statements and `import()` expressions.

```
// my-app.js, treated as an ES module 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 "type" value 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 the nearest parent `package.json`.

Files ending with `.cjs` are always loaded as CommonJS regardless of the nearest parent `package.json`.

```
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 can be used to mix types within the same package:

- Within a `"type": "module"` package, 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).
- Within a `"type": "commonjs"` package, 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).

`--input-type` flag

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

```
node --input-type=module --eval "import { sep } from 'path'; console.log(sep);"

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

Determining package manager

> *Stability: 1 - Experimental*

While all Node.js projects are expected to be installable by all package managers once published, their development teams are often required to use one specific package manager. To make this process easier, Node.js ships with a tool called Corepack that aims to make all package managers transparently available in your environment - provided you have Node.js installed.

By default Corepack won't enforce any specific package manager and will use the generic "Last Known Good" versions associated with each Node.js release, but you can improve this experience by setting the `"packageManager"` field in your project's `package.json`.

Package entry points

In a package's `package.json` file, two fields can define entry points for a package: `"main"` and `"exports"`. The `"main"` field is supported in all versions of Node.js, but its capabilities are limited: it only defines the main entry point of the package.

The `"exports"` field provides an alternative to `"main"` where the package main entry point can be defined while also encapsulating the package, **preventing any other entry points besides those defined in `"exports"`**. This encapsulation allows module authors to define a public interface for their package.

If both `"exports"` and `"main"` are defined, the `"exports"` field takes precedence over `"main"`. `"exports"` are not specific to ES modules or CommonJS; `"main"` is overridden by `"exports"` if it exists. As such `"main"` cannot be used as a fallback for CommonJS but it can be used as a fallback for legacy versions of Node.js that do not support the `"exports"` field.

Conditional exports can be used within `"exports"` to define different package entry points per environment, including whether the package is referenced via `require` or via `import`. For more information about supporting both CommonJS and ES Modules in a single package please consult the dual CommonJS/ES module packages section.

Warning: Introducing the `"exports"` field prevents consumers of a package from using any entry points that are not defined, including the `package.json` (e.g. `require('your-package/package.json')`. **This will likely be a breaking change.**

To make the introduction of `"exports"` non-breaking, ensure that every previously supported entry point is exported. It is best to explicitly specify entry points so that the package's public API is well-defined. For example, a project that previous exported `main`, `lib`, `feature`, and the `package.json` could use the following `package.exports`:

```
{
  "name": "my-mod",
  "exports": {
    ".": "./lib/index.js",
    "./lib": "./lib/index.js",
    "./lib/index": "./lib/index.js",
    "./lib/index.js": "./lib/index.js",
    "./feature": "./feature/index.js",
    "./feature/index.js": "./feature/index.js",
    "./package.json": "./package.json"
  }
}
```

Alternatively a project could choose to export entire folders:

```
{
  "name": "my-mod",
  "exports": {
    ".": "./lib/index.js",
    "./lib": "./lib/index.js",
    "./lib/*": "./lib/*.js",
    "./feature": "./feature/index.js",
    "./feature/*": "./feature/*.js",
    "./package.json": "./package.json"
```

```
  }
}
```

As a last resort, package encapsulation can be disabled entirely by creating an export for the root of the package `"./*": "./*"`. This exposes every file in the package at the cost of disabling the encapsulation and potential tooling benefits this provides. As the ES Module loader in Node.js enforces the use of the full specifier path, exporting the root rather than being explicit about entry is less expressive than either of the prior examples. Not only is encapsulation lost but module consumers are unable to `import feature from 'my-mod/feature'` as they need to provide the full path `import feature from 'my-mod/feature/index.js`.

Main entry point export

To set the main entry point for a package, it is advisable to define both `"exports"` and `"main"` in the package's `package.json` file:

```
{
  "main": "./main.js",
  "exports": "./main.js"
}
```

When the `"exports"` field is defined, all subpaths of the package are encapsulated and no longer available to importers. For example, `require('pkg/subpath.js')` throws an `ERR_PACKAGE_PATH_NOT_EXPORTED` error.

This encapsulation of exports provides more reliable guarantees about package interfaces for tools and when handling semver upgrades for a package. It is not a strong encapsulation since a direct require of any absolute subpath of the package such as `require('/path/to/node_modules/pkg/subpath.js')` will still load `subpath.js`.

Subpath exports

When using the `"exports"` field, custom subpaths can be defined along with the main entry point by treating the main entry point as the `"."` subpath:

```
{
  "main": "./main.js",
  "exports": {
    ".": "./main.js",
    "./submodule": "./src/submodule.js"
  }
}
```

Now only the defined subpath in `"exports"` can be imported by a consumer:

```
import submodule from 'es-module-package/submodule';
// Loads ./node_modules/es-module-package/src/submodule.js
```

While other subpaths will error:

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

Subpath imports

In addition to the `"exports"` field, it is possible to define internal package import maps that only apply to import specifiers from within the package itself.

Entries in the imports field must always start with `#` to ensure they are disambiguated from package specifiers.

For example, the imports field can be used to gain the benefits of conditional exports for internal modules:

```
// package.json
{
  "imports": {
    "#dep": {
      "node": "dep-node-native",
      "default": "./dep-polyfill.js"
    }
  },
  "dependencies": {
    "dep-node-native": "^1.0.0"
  }
}
```

where `import '#dep'` does not get the resolution of the external package `dep-node-native` (including its exports in turn), and instead gets the local file `./dep-polyfill.js` relative to the package in other environments.

Unlike the `"exports"` field, the `"imports"` field permits mapping to external packages.

The resolution rules for the imports field are otherwise analogous to the exports field.

Subpath patterns

For packages with a small number of exports or imports, we recommend explicitly listing each exports subpath entry. But for packages that have large numbers of subpaths, this might cause `package.json` bloat and maintenance issues.

For these use cases, subpath export patterns can be used instead:

```
// ./node_modules/es-module-package/package.json
{
  "exports": {
    "./features/*": "./src/features/*.js"
  },
  "imports": {
    "#internal/*": "./src/internal/*.js"
  }
}
```

`*` maps expose nested subpaths as it is a string replacement syntax only.

All instances of `*` on the right hand side will then be replaced with this value, including if it contains any `/` separators.

```
import featureX from 'es-module-package/features/x';
// Loads ./node_modules/es-module-package/src/features/x.js

import featureY from 'es-module-package/features/y/y';
// Loads ./node_modules/es-module-package/src/features/y/y.js

import internalZ from '#internal/z';
// Loads ./node_modules/es-module-package/src/internal/z.js
```

This is a direct static replacement without any special handling for file extensions. In the previous example, `pkg/features/x.json` would be resolved to `./src/features/x.json.js` in the mapping.

The property of exports being statically enumerable is maintained with exports patterns since the individual exports for a package can be determined by treating the right hand side target pattern as a `**` glob against the list of files within the package. Because `node_modules` paths are forbidden in exports targets, this expansion is dependent on only the files of the package itself.

To exclude private subfolders from patterns, `null` targets can be used:

```
// ./node_modules/es-module-package/package.json
{
  "exports": {
    "./features/*": "./src/features/*.js",
    "./features/private-internal/*": null
  }
}
```

```
import featureInternal from 'es-module-package/features/private-internal/m';
// Throws: ERR_PACKAGE_PATH_NOT_EXPORTED

import featureX from 'es-module-package/features/x';
// Loads ./node_modules/es-module-package/src/features/x.js
```

Exports sugar

If the `"."` export is the only export, the `"exports"` field provides sugar for this case being the direct `"exports"` field value.

If the `"."` export has a fallback array or string value, then the `"exports"` field can be set to this value directly.

```
{
  "exports": {
    ".": "./main.js"
  }
}
```

can be written:

```
{
  "exports": "./main.js"
}
```

Conditional exports

Conditional exports provide a way to map to different paths depending on certain conditions. They are supported for both CommonJS and ES module imports.

For example, a package that wants to provide different ES module exports for `require()` and `import` can be written:

```
// package.json
{
  "main": "./main-require.cjs",
  "exports": {
    "import": "./main-module.js",
    "require": "./main-require.cjs"
  },
  "type": "module"
}
```

Node.js implements the following conditions, listed in order from most specific to least specific as conditions should be defined:

- `"node-addons"` - similar to `"node"` and matches for any Node.js environment. This condition can be used to provide an entry point which uses native C++ addons as opposed to an entry point which is more universal and doesn't rely on native addons. This condition can be disabled via the `--no-addons` flag.
- `"node"` - matches for any Node.js environment. Can be a CommonJS or ES module file. *In most cases explicitly calling out the Node.js platform is not necessary.*
- `"import"` - matches when the package is loaded via `import` or `import()`, or via any top-level import or resolve operation by the ECMAScript module loader. Applies regardless of the module format of the target file. *Always mutually exclusive with* `"require"`.
- `"require"` - matches when the package is loaded via `require()`. The referenced file should be loadable with `require()` although the condition matches regardless of the module format of the target file. Expected formats include CommonJS, JSON, and native addons but not ES modules as `require()` doesn't support them. *Always mutually exclusive with* `"import"`.
- `"default"` - the generic fallback that always matches. Can be a CommonJS or ES module file. *This condition should always come last.*

Within the `"exports"` object, key order is significant. During condition matching, earlier entries have higher priority and take precedence over later entries. *The general rule is that conditions should be from most specific to least specific in object order*.

Using the `"import"` and `"require"` conditions can lead to some hazards, which are further explained in the dual CommonJS/ES module packages section.

The `"node-addons"` condition can be used to provide an entry point which uses native C++ addons. However, this condition can be disabled via the `--no-addons` flag. When using `"node-addons"`, it's recommended to treat `"default"` as an enhancement that provides a more universal entry point, e.g. using WebAssembly instead of a native addon.

Conditional exports can also be extended to exports subpaths, for example:

```
{
  "main": "./main.js",
  "exports": {
    ".": "./main.js",
    "./feature": {
      "node": "./feature-node.js",
      "default": "./feature.js"
    }
  }
}
```

Defines a package where `require('pkg/feature')` and `import 'pkg/feature'` could provide different implementations between Node.js and other JS environments.

When using environment branches, always include a `"default"` condition where possible. Providing a `"default"` condition ensures that any unknown JS environments are able to use this universal implementation, which helps avoid these JS environments from having to pretend to be existing environments in order to support packages with conditional exports. For this reason, using `"node"` and `"default"` condition branches is usually preferable to using `"node"` and `"browser"` condition branches.

Nested conditions

In addition to direct mappings, Node.js also supports nested condition objects.

For example, to define a package that only has dual mode entry points for use in Node.js but not the browser:

```
{
  "main": "./main.js",
  "exports": {
    "node": {
      "import": "./feature-node.mjs",
      "require": "./feature-node.cjs"
    },
    "default": "./feature.mjs"
  }
}
```

Conditions continue to be matched in order as with flat conditions. If a nested condition does not have any mapping it will continue checking the remaining conditions of the parent condition. In this way nested conditions behave analogously to nested JavaScript `if` statements.

Resolving user conditions

When running Node.js, custom user conditions can be added with the `--conditions` flag:

```
node --conditions=development main.js
```

which would then resolve the `"development"` condition in package imports and exports, while resolving the existing `"node"`, `"node-addons"`, `"default"`, `"import"`, and `"require"` conditions as appropriate.

Any number of custom conditions can be set with repeat flags.

Community Conditions Definitions

Condition strings other than the `"import"`, `"require"`, `"node"`, `"node-addons"` and `"default"` conditions implemented in Node.js core are ignored by default.

Other platforms may implement other conditions and user conditions can be enabled in Node.js via the `--conditions` / `-C` flag.

Since custom package conditions require clear definitions to ensure correct usage, a list of common known package conditions and their strict definitions is provided below to assist with ecosystem coordination.

- `"types"` - can be used by typing systems to resolve the typing file for the given export. *This condition should always be included first.*
- `"deno"` - indicates a variation for the Deno platform.
- `"browser"` - any web browser environment.
- `"development"` - can be used to define a development-only environment entry point, for example to provide additional debugging context such as better error messages when running in a development mode. *Must always be mutually exclusive with* `"production"`.
- `"production"` - can be used to define a production environment entry point. *Must always be mutually exclusive with* `"development"`.

New conditions definitions may be added to this list by creating a pull request to the Node.js documentation for this section. The requirements for listing a new condition definition here are that:

- The definition should be clear and unambiguous for all implementers.
- The use case for why the condition is needed should be clearly justified.
- There should exist sufficient existing implementation usage.
- The condition name should not conflict with another condition definition or condition in wide usage.
- The listing of the condition definition should provide a coordination benefit to the ecosystem that wouldn't otherwise be possible. For example, this would not necessarily be the case for company-specific or application-specific conditions.

The above definitions may be moved to a dedicated conditions registry in due course.

Self-referencing a package using its name

Within a package, the values defined in the package's `package.json` `"exports"` field can be referenced via the package's name. For example, assuming the `package.json` is:

```
// package.json
{
  "name": "a-package",
  "exports": {
    ".": "./main.mjs",
    "./foo": "./foo.js"
```

```
  }
}
```

Then any module *in that package* can reference an export in the package itself:

```
// ./a-module.mjs
import { something } from 'a-package'; // Imports "something" from ./main.mjs.
```

Self-referencing is available only if `package.json` has `"exports"`, and will allow importing only what that `"exports"` (in the `package.json`) allows. So the code below, given the previous package, will generate a runtime error:

```
// ./another-module.mjs

// Imports "another" from ./m.mjs. Fails because
// the "package.json" "exports" field
// does not provide an export named "./m.mjs".
import { another } from 'a-package/m.mjs';
```

Self-referencing is also available when using `require`, both in an ES module, and in a CommonJS one. For example, this code will also work:

```
// ./a-module.js
const { something } = require('a-package/foo'); // Loads from ./foo.js.
```

Finally, self-referencing also works with scoped packages. For example, this code will also work:

```
// package.json
{
  "name": "@my/package",
  "exports": "./index.js"
}
```

```
// ./index.js
module.exports = 42;
```

```
// ./other.js
console.log(require('@my/package'));
```

```
$ node other.js
42
```

Dual CommonJS/ES module packages

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) the top-level `"module"` field.

Node.js can now run ES module entry points, and a package can contain both CommonJS and ES module entry points (either via separate specifiers such as `'pkg'` and `'pkg/es-module'`, or both at the same specifier via Conditional exports). Unlike in the scenario where `"module"` is only used by bundlers, or ES module files are transpiled into CommonJS on the fly before evaluation by Node.js, the files referenced by the ES module entry point are evaluated as ES modules.

Dual package hazard

When an application is using a package that provides both CommonJS and ES module sources, there is a risk of certain bugs if both versions of the package get loaded. This potential comes from the fact that the `pkgInstance` created by `const pkgInstance = require('pkg')` is not the same as the `pkgInstance` created by `import pkgInstance from 'pkg'` (or an alternative main path like `'pkg/module'`). This is the "dual package hazard," where two versions of the same package can be loaded within the same runtime environment. While it is unlikely that an application or package would intentionally load both versions directly, it is common for an application to load one version while a dependency of the application loads the other version. This hazard can happen because Node.js supports intermixing CommonJS and ES modules, and can lead to unexpected behavior.

If the package main export is a constructor, an `instanceof` comparison of instances created by the two versions returns `false`, and if the export is an object, properties added to one (like `pkgInstance.foo = 3`) are not present on the other. This differs from how `import` and `require` statements work in all-CommonJS or all-ES module environments, respectively, and therefore is surprising to users. It also differs from the behavior users are familiar with when using transpilation via tools like Babel or `esm`.

Writing dual packages while avoiding or minimizing hazards

First, the hazard described in the previous section occurs when a package contains both CommonJS and ES module sources and both sources are provided for use in Node.js, either via separate main entry points or exported paths. A package might instead be written where any version of Node.js receives only CommonJS sources, and any separate ES module sources the package might contain are intended only for other environments such as browsers. Such a package would be usable by any version of Node.js, since `import` can refer to CommonJS files; but it would not provide any of the advantages of using ES module syntax.

A package might also switch from CommonJS to ES module syntax in a breaking change version bump. This has the disadvantage that the newest version of the package would only be usable in ES module-supporting versions of Node.js.

Every pattern has tradeoffs, but there are two broad approaches that satisfy the following conditions:

1. The package is usable via both `require` and `import`.
2. The package is usable in both current Node.js and older versions of Node.js that lack support for ES modules.
3. The package main entry point, e.g. `'pkg'` can be used by both `require` to resolve to a CommonJS file and by `import` to resolve to an ES module file. (And likewise for exported paths, e.g. `'pkg/feature'`.)
4. The package provides named exports, e.g. `import { name } from 'pkg'` rather than `import pkg from 'pkg'; pkg.name`.
5. The package is potentially usable in other ES module environments such as browsers.
6. The hazards described in the previous section are avoided or minimized.

Approach #1: Use an ES module wrapper

Write the package in CommonJS or transpile ES module sources into CommonJS, and create an ES module wrapper file that defines the named exports. Using Conditional exports, the ES module wrapper is used for `import` and the CommonJS entry point for `require`.

```
// ./node_modules/pkg/package.json
{
  "type": "module",
  "main": "./index.cjs",
  "exports": {
    "import": "./wrapper.mjs",
    "require": "./index.cjs"
  }
}
```

The preceding example uses explicit extensions `.mjs` and `.cjs`. If your files use the `.js` extension, `"type": "module"` will cause such files to be treated as ES modules, just as `"type": "commonjs"` would cause them to be treated as CommonJS. See Enabling.

```
// ./node_modules/pkg/index.cjs
exports.name = 'value';
```

```
// ./node_modules/pkg/wrapper.mjs
import cjsModule from './index.cjs';
export const name = cjsModule.name;
```

In this example, the `name` from `import { name } from 'pkg'` is the same singleton as the `name` from `const { name } = require('pkg')`. Therefore `===` returns `true` when comparing the two `name`s and the divergent specifier hazard is avoided.

If the module is not simply a list of named exports, but rather contains a unique function or object export like `module.exports = function () { ... }`, or if support in the wrapper for the `import pkg from 'pkg'` pattern is desired, then the wrapper would instead be written to export the default optionally along with any named exports as well:

```
import cjsModule from './index.cjs';
export const name = cjsModule.name;
export default cjsModule;
```

This approach is appropriate for any of the following use cases:

- The package is currently written in CommonJS and the author would prefer not to refactor it into ES module syntax, but wishes to provide named exports for ES module consumers.
- The package has other packages that depend on it, and the end user might install both this package and those other packages. For example a `utilities` package is used directly in an application, and a `utilities-plus` package adds a few more functions to `utilities`. Because the wrapper exports underlying CommonJS files, it doesn't matter if `utilities-plus` is written in CommonJS or ES module syntax; it will work either way.

- The package stores internal state, and the package author would prefer not to refactor the package to isolate its state management. See the next section.

A variant of this approach not requiring conditional exports for consumers could be to add an export, e.g. `"./module"`, to point to an all-ES module-syntax version of the package. This could be used via `import 'pkg/module'` by users who are certain that the CommonJS version will not be loaded anywhere in the application, such as by dependencies; or if the CommonJS version can be loaded but doesn't affect the ES module version (for example, because the package is stateless):

```
// ./node_modules/pkg/package.json
{
  "type": "module",
  "main": "./index.cjs",
  "exports": {
    ".": "./index.cjs",
    "./module": "./wrapper.mjs"
  }
}
```

Approach #2: Isolate state

A `package.json` file can define the separate CommonJS and ES module entry points directly:

```
// ./node_modules/pkg/package.json
{
  "type": "module",
  "main": "./index.cjs",
  "exports": {
    "import": "./index.mjs",
    "require": "./index.cjs"
  }
}
```

This can be done if both the CommonJS and ES module versions of the package are equivalent, for example because one is the transpiled output of the other; and the package's management of state is carefully isolated (or the package is stateless).

The reason that state is an issue is because both the CommonJS and ES module versions of the package might get used within an application; for example, the user's application code could `import` the ES module version while a dependency `require`s the CommonJS version. If that were to occur, two copies of the package would be loaded in memory and therefore two separate states would be present. This would likely cause hard-to-troubleshoot bugs.

Aside from writing a stateless package (if JavaScript's `Math` were a package, for example, it would be stateless as all of its methods are static), there are some ways to isolate state so that it's shared between the potentially loaded CommonJS and ES module instances of the package:

1. If possible, contain all state within an instantiated object. JavaScript's `Date`, for example, needs to be instantiated to contain state; if it were a package, it would be used like this:

   ```
   import Date from 'date';
   const someDate = new Date();
   ```

```
// someDate contains state; Date does not
```

The `new` keyword isn't required; a package's function can return a new object, or modify a passed-in object, to keep the state external to the package.

2. Isolate the state in one or more CommonJS files that are shared between the CommonJS and ES module versions of the package. For example, if the CommonJS and ES module entry points are `index.cjs` and `index.mjs`, respectively:

```
// ./node_modules/pkg/index.cjs
const state = require('./state.cjs');
module.exports.state = state;
```

```
// ./node_modules/pkg/index.mjs
import state from './state.cjs';
export {
  state
};
```

Even if `pkg` is used via both `require` and `import` in an application (for example, via `import` in application code and via `require` by a dependency) each reference of `pkg` will contain the same state; and modifying that state from either module system will apply to both.

Any plugins that attach to the package's singleton would need to separately attach to both the CommonJS and ES module singletons.

This approach is appropriate for any of the following use cases:

- The package is currently written in ES module syntax and the package author wants that version to be used wherever such syntax is supported.
- The package is stateless or its state can be isolated without too much difficulty.
- The package is unlikely to have other public packages that depend on it, or if it does, the package is stateless or has state that need not be shared between dependencies or with the overall application.

Even with isolated state, there is still the cost of possible extra code execution between the CommonJS and ES module versions of a package.

As with the previous approach, a variant of this approach not requiring conditional exports for consumers could be to add an export, e.g. `"./module"`, to point to an all-ES module-syntax version of the package:

```
// ./node_modules/pkg/package.json
{
  "type": "module",
  "main": "./index.cjs",
  "exports": {
    ".": "./index.cjs",
    "./module": "./index.mjs"
  }
}
```

Node.js `package.json` field definitions

This section describes the fields used by the Node.js runtime. Other tools (such as npm) use additional fields which are ignored by Node.js and not documented here.

The following fields in `package.json` files are used in Node.js:

- `"name"` - Relevant when using named imports within a package. Also used by package managers as the name of the package.
- `"main"` - The default module when loading the package, if exports is not specified, and in versions of Node.js prior to the introduction of exports.
- `"packageManager"` - The package manager recommended when contributing to the package. Leveraged by the Corepack shims.
- `"type"` - The package type determining whether to load `.js` files as CommonJS or ES modules.
- `"exports"` - Package exports and conditional exports. When present, limits which submodules can be loaded from within the package.
- `"imports"` - Package imports, for use by modules within the package itself.

`"name"`

- Type: {string}

```
{
  "name": "package-name"
}
```

The `"name"` field defines your package's name. Publishing to the *npm* registry requires a name that satisfies certain requirements.

The `"name"` field can be used in addition to the `"exports"` field to self-reference a package using its name.

`"main"`

- Type: {string}

```
{
  "main": "./main.js"
}
```

The `"main"` field defines the entry point of a package when imported by name via a `node_modules` lookup. Its value is a path.

When a package has an `"exports"` field, this will take precedence over the `"main"` field when importing the package by name.

It also defines the script that is used when the package directory is loaded via `require()`.

```
require('./path/to/directory'); // This resolves to ./path/to/directory/main.js.
```

`"packageManager"`

> *Stability: 1 - Experimental*

- Type: {string}

```
{
  "packageManager": "<package manager name>@<version>"
}
```

The `"packageManager"` field defines which package manager is expected to be used when working on the current project. It can set to any of the supported package managers, and will ensure that your teams use the exact same package manager versions without having to install anything else than Node.js.

This field is currently experimental and needs to be opted-in; check the Corepack page for details about the procedure.

`"type"`

- Type: {string}

The `"type"` field defines the module format that Node.js uses for all `.js` files that have that `package.json` file as their nearest parent.

Files ending with `.js` are 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 a node_modules folder or the volume root is reached.

```
// package.json
{
  "type": "module"
}
```

```
# In same folder as preceding package.json
node my-app.js # Runs as ES module
```

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

`import` statements of `.js` 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
```

Regardless of the value of the `"type"` field, `.mjs` files are always treated as ES modules and `.cjs` files are always treated as CommonJS.

`"exports"`

- Type: {Object} | {string} | {string[]}

```
{
  "exports": "./index.js"
}
```

The `"exports"` field allows defining the entry points of a package when imported by name loaded either via a `node_modules` lookup or a self-reference to its own name. It is supported in Node.js 12+ as an alternative to the `"main"` that can support defining subpath exports and conditional exports while encapsulating internal unexported modules.

Conditional Exports can also be used within `"exports"` to define different package entry points per environment, including whether the package is referenced via `require` or via `import`.

All paths defined in the `"exports"` must be relative file URLs starting with `./`.

`"imports"`

- Type: {Object}

```
// package.json
{
  "imports": {
    "#dep": {
      "node": "dep-node-native",
      "default": "./dep-polyfill.js"
    }
  },
  "dependencies": {
    "dep-node-native": "^1.0.0"
  }
}
```

Entries in the imports field must be strings starting with `#`.

Import maps permit mapping to external packages.

This field defines subpath imports for the current package.