Module Specifier Mapping in Node.js

Previously: Bare Module Specifier Resolution in node.js
Contributors: Guy Bedford, Geoffrey Booth, Jan Krems, Saleh Abdel Motaal

Motivating Examples

- A package (react-dom) has a dedicated entrypoint react-dom/server for code that isn't compatible with a browser environment.
- A package (angular) exposes multiple independent APIs, modeled via import paths like angular/common/http.
- A package (lodash) allows to import individual functions, e.g. lodash/map.
- A package is exclusively exposing an ESM interface.
- A package is exclusively exposing a CJS interface.
- A package is exposing both an ESM and a CJS interface.
- A project wants to mix both ESM and CJS code, with CJS running as part of the ESM module graph.
- A package wants to expose multiple entry points as its public API without leaking internal directory structure.

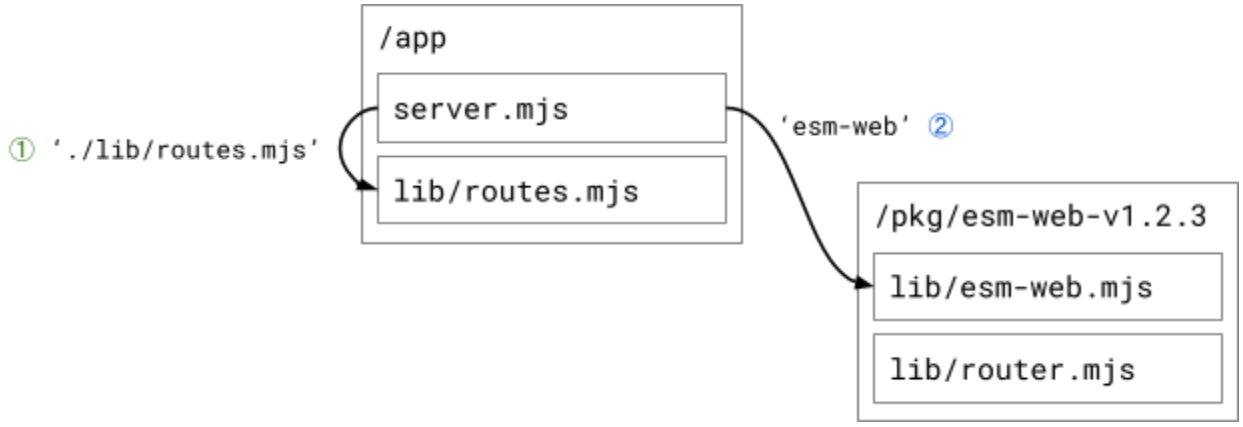
Use Cases

There are three high-level reasons for wanting dynamic specifier resolution:

1. Support different versions of a dependency without knowledge of how they will installed or what their internal directory structure is. Files in a project can `import ’lodash’` and won’t have to change with every release or refactor of lodash.
2. Support platform or environment difference. Code may have specific references to DOM-, react-native, electron, or node APIs. There may also be different versions for development vs. production use. This is handled today using dynamic require based on runtime checks or via the browser/react-native field in package.json.
3. Support “vanity” specifiers that look like plain relative paths but are interpreted based on a set of rules. E.g. in CommonJS, a specifier `./models` may resolve to `./models.js`, `models/index.js`, `./models.json`, or any number of other target URLs depending on what files and directories exist. This automatic searching algorithm is not present in browser ESM implementations, but there is still a desire for specifiers that don’t match paths and filenames on disk. The goal is to support such specifiers in ESM in node.js, but through explicit configuration rather than a CommonJS-style automatic algorithm.

Overview

This proposal assumes a JavaScript project setup that is widespread in node.js and in packages distributed via npm: Each project is authored within a self-contained directory. Files within the project reference each other using relative paths①. We'll call those **internal specifiers**. These specifiers are usually expected to resolve to files within the same project.
When a project needs access to files in another project, it uses a symbolic string② instead of a relative path. The final location of the loaded module may and often does live outside of the original directory. We'll call those **external specifiers** in this proposal.



One observation made in the import maps proposal is that we can express the entire specifier resolution using a single data structure: A table that, based on the URL prefix of the file triggering the import, maps input specifiers to fully resolved URLs. Assuming the example above describes files on disk, we could write such a **resolution table**. Top-level keys are URL prefixes and each contains a map from input specifier to output URL:

"file:///app/":
  "file:///app/lib/routes.mjs": "file:///app/lib/routes.mjs" ③
  "esm-web": "file:///pkg/esm-web-v1.2.3/lib/esm-web.mjs"
"file:///pkg/esm-web-v1.2.3/":
  "file:///pkg/esm-web-v1.2.3/lib/router.mjs": "file:///pkg/esm-web-v1.2.3/lib/router.mjs" ③

A way to read this data structure is: "In any module whose URL begins with file:///app/, we'd resolve 'esm-web' to file:///pkg/esm-web-v1.2.3/lib/esm-web.mjs".

One thing to call out is that there's a fundamental difference between how internal and external specifiers look in this resolution table: For internal specifiers, the input and the target are both inside of the same project. For external specifiers, the target appears in the URL prefix of the consuming project instead.

There are also trivial resolutions③ that fall out from how relative URL resolution works. If we'd include the raw relative specifiers inside of the table, the table would have to include every possible traversal path. That's why we normalize the specifiers by doing relative resolution first and then look up any special mappings. For brevity's sake we'll omit entries where the normalized input specifier matches the target URL as it does in the example above.

Proposed Semantics

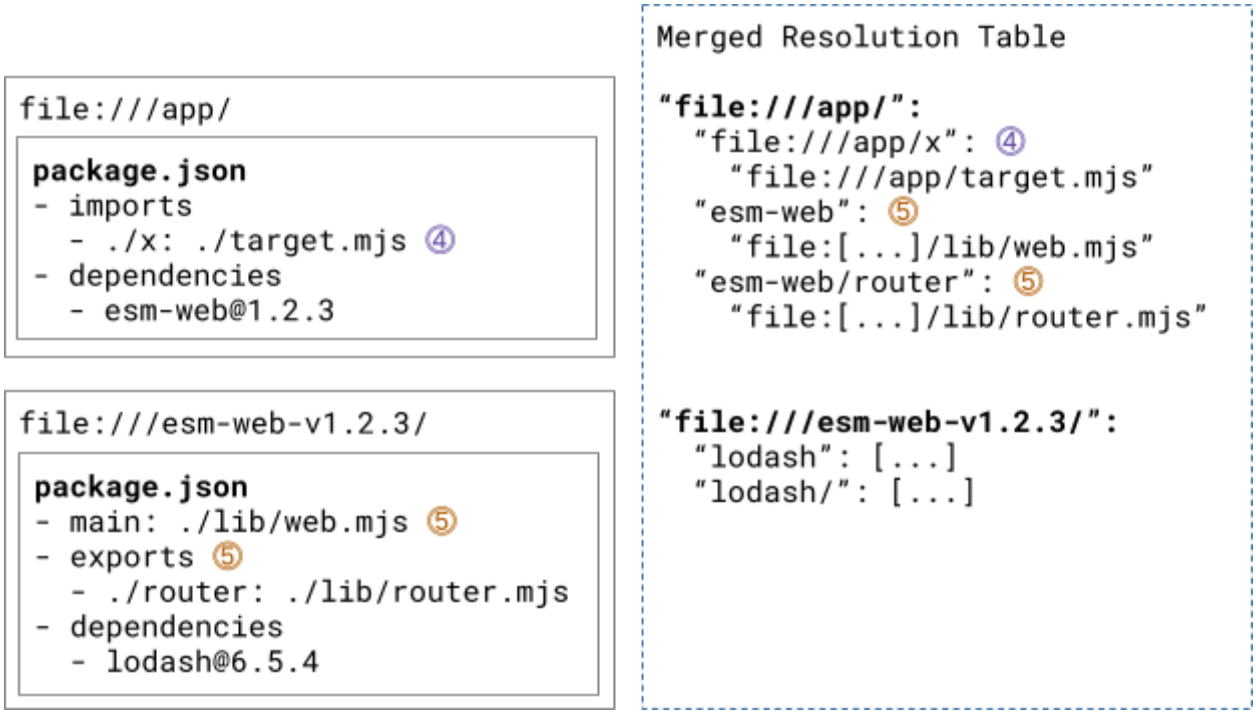
Throughout the description of semantics, we'll use the concept of a resolution table established above. That doesn't mean that every implementation will literally build a table or that it would be this exact data structure. The implementation in node.js might not use an in-memory data structure similar to a resolution table. But the behavior should match a system that did.

package.json

We propose two new fields for package.json, one for each kind of specifier:

- `imports` for mapping **internal specifiers**. They apply to the current package boundary.
- `exports` (combined with the existing `main` field) for mapping **external specifiers**. They apply to each dependent package boundary.

The interpretation of those fields depends on the concept of a **package.json boundary**. Each of these boundaries creates a URL prefix for the resolution table. The `imports` entries are added to the same URL prefix in which they appear.④ The `exports` and `main` entries are added to the URL prefix of each dependent package instead.⑤ In the case of a collision between own `imports` and `exports` from a dependency, the `imports` take precedent.



Note: The dependencies field is given for illustrative purposes. In node's implementation, the dependency relationship would most likely be established by searching node_modules directories. But other loaders (tink, yarn) may use their own mechanisms like explicit metadata created during installation.

Building the Resolution Table

So far we've given examples of resolution tables but haven't really explained how it is constructed.

1. For each package boundary identified by `urlPrefix`:
    a. Create an empty table for `urlPrefix`.
    b. [imports] For each `imports` entry `[specifier, target]` in `${urlPrefix}package.json`:
        i. If specifier starts with `/`, fail with "invalid imports".
        ii. If specifier starts with `.`:
            1. Set `specifier` to `new URL(specifier, urlPrefix).href`.
            2. Assert: `specifier` starts with `urlPrefix`.
        iii. If target starts with `/`, fails with "invalid imports".
        iv. If target starts with `.`:
            1. Set `target` to `new URL(target, urlPrefix).href`.
            2. Assert: `target` starts with `urlPrefix`.
        v. Assert: target is a valid URL.

        vi. Add `specifier => target` to the table for `urlPrefix` if the table doesn't contain this key yet.
    c. [exports] For each dependency with `depName` and `depPrefix`:
        i. Set `exports` to the `exports` field in `${depPrefix}package.json` or `{ "./": "./" }` if it isn't present.
        ii. If a `main` field exists, set `exports["."]` to its value. Otherwise, assert that `exports["."]` is not set. This ensures that the primary export can only be set via the `main` field.
        iii. For each `exports` entry `[specifierPostfix, target]`:
            1. If `specifierPostfix` isn't "." and doesn't start with "./" or contains a dot (".") after a slash ("/"), fail with "invalid exports". This could potentially be relaxed in the future to allow `./.htaccess` as a valid exports entry.
            2. Set specifier to `${depName}${specifierPostfix.substr(1)`.
            3. If target starts with `/`, fails with "invalid imports".
            4. If target starts with `.`:
                a. Set `target` to `new URL(target, depPrefix).href`.
                b. Assert: `target` starts with `depPrefix`.
            5. Assert: target is a valid URL.
            6. Add `specifier => target` to the table for `urlPrefix` if the table doesn't contain this key yet.

This algorithm is designed to give local data precedent over remote data. E.g. own `imports` win over the `exports` loaded from a dependency.

Resolution Table Lookup

Rough draft / placeholder inspired by the existing resolution and discussions for `main`:

1. Start with a specifier string `specifier` and a referring module URL `referrerURL`.
2. If `specifier` is a relative specifier, resolve it to an absolute URL using `referrerURL`. **Note:** `specifier` is now either an absolute URL or a bare specifier.
3. Find the longest url prefix `urlPrefix` and its associated `table` for `referrerURL`. If a match is found:
    a. If `table` contains an exact match for `specifier`, set `specifier` to `table[specifier]`.
    b. Otherwise, find the longest `prefixKey` in `table` that is a prefix of `specifier` and ends with a `/`.
    c. If `prefixKey` has been found, remove the prefix from `specifier` and set `specifier` to the result of appending the remainder of `specifier` to `table[prefixKey]`.
4. **Assert:** `specifier` is an absolute URL.

5. For CommonJS compat, if the specifier protocol is file://, check if a file at the exact URL exists. If it doesn't, yield to require resolution (minus potential resolution table checks). If a require resolution exists, **assert** that the resulting URL is interpreted as CommonJS. Set specifier to the resulting URL.
   Note: This ensures that `main` fields of CommonJS packages work as expected while web-compatible code in modules doesn't introduce accidental web-incompatibilities.
6. Return `specifier` as the final URL.

TBD: The algorithm to interpret a specifier given a table.
TBD: The algorithm to honor the table in import, including CJS resolution fallback.

Resolution Table in Require

Rough draft:

1. Do basic resolution to absolute URL of relative specifiers.
2. Check resolution table based on referring module URL. Only entries with file URLs will be relevant.
3. Run the same logic as for import, only that the CommonJS compat is implied (non-file isn't allowed) and that the assertion of file format at the end is skipped.

TBD: The algorithm to honor the table in require, including CJS resolution fallback.

Challenge: We want to prevent a duplicate lookup when falling back for specifiers that cannot be resolved in import.

- "imports" defines how specifiers resolve *within* this package. This is akin to the import maps proposal, but scoped to this package. This allows code like `'import fetch from 'fetch'`, where 'fetch' can be defined as a built-in library (if available) or a dependency otherwise. You could also write e.g. `import express from 'https://unpkg.com/pkg@4.x'` in your package, where "imports" maps 'https://unpkg.com/pkg@4.x' to a local dependency in Node, while in a browser context the URL would be used as is. Other specifiers could be mapped to varying destinations based on environment (browser vs. Node, Linux vs Windows, etc.).

"exports" public API

Here's a complete package.json example, for a hypothetical module named @momentjs/moment:

```
{
  "name": "@momentjs/moment",
  "version": "0.0.0",
  "type": "module",
```

```
  "main": "./dist/moment.js",
  "exports": {
    "/": "./dist/util/",
    "/timezones/": "./data/timezones/",
    "/timezones/utc": "./data/timezones/utc/index.mjs"
  }
}
```

Within the "exports" object, the key string is concatenated after the name field, e.g. import utc from '@momentjs/moment/timezones/utc' is formed from '@momentjs/moment' + '/timezones/utc'. Note that this is string manipulation, not a file path: "/timezones/utc" is allowed, but just "timezones/utc" is not.

Keys that end in slashes can map to folder roots, following the pattern in the browser import maps proposal: "/timezones/": "/data/timezones/" would allow import pdt from "@momentjs/moment/timezones/pdt.mjs" to import ./data/timezones/pdt.mjs.

- There is no way to map the root; that's what "main" is for. "exports" can only be used for paths within the package.
- Mapping a key of "/" to a value of "./" exposes all files in the package, where "/": "./" would allow import privateHelpers from "@momentjs/moment/private-helpers.mjs" to import ./private-helpers.mjs.
- When mapping to a folder, both the left and right sides must end in slashes: "/": "./dist/", not "/": "./dist".
- Unlike in CommonJS, there is no automatic searching for index.js or index.mjs or for file extensions. This matches the behavior of the import maps proposal and the default behavior in Node 12's --experimental-modules.

The value of an export, e.g. "./src/moment.mjs", must begin with . to signify a relative path (e.g. "./src" is okay, but "/src" or "src" are not). This is to reserve potential future use for "exports" to export things referenced via specifiers that aren't relatively-resolved files, such as other packages or other protocols.

There is the potential for collisions in the exports, such as "./timezones/" and "./timezones/utc" in the example above (e.g. if there's a file named utc in the ./data/timezones folder). Rough outline of a possible resolution algorithm:

1. Find the package matching the base specifier, e.g. @momentjs/moment or request.
2. Load its exports map.
3. If there is an exact match for the requested specifier, return the resolution.
4. Otherwise, find the longest matching path prefix. If there is a path prefix, return the resolution by applying the prefix.
5. Return an error - no mapping found.

In the future, the algorithm might be adjusted to align with work done in the import maps proposal.

“imports” specifier resolution within the package

TBD

References

Prior Art

- package.json#browser
- Import Maps
- package.json#mimes
- node.js ESM resolver spec

[TMP] Raw Notes

```js
{
  "imports": {
    "graceful-fs": "?", // package.json#main
    "graceful-fs/no-side-effects": "?" // package.json#exports
  },
  "scopes": [
    {
      "prefix": "fs://project/node_modules/graceful-fs",
      "imports": { /* ? */ }, // package.json#imports
    },
    {
      "prefix": "fs://project/node_modules/gofer",
      "imports": {
        "./lib/request.js": [
          "./lib/request.node.js",
          // or: { node: true, value: "./lib/request.node.js" },
          "./lib/request.browser.js",
          // or: { browser: true, value: "./lib/request.node.js" },
          "./lib/request.react-native.js",
          "./lib/request.electron.js",
        ],
      },
    },
  ]
}
```

```js
{
  "main": "./old-common.js",
  "exports": {
    "/": "/",
    // vanity support for nicer import specifier
    "/entry": "/entry.js"
  }
}
```

Outstanding Issues

* How do we prevent a "same specifier, different instance" issue.
  Can we prevent it?
  Can we discourage it?
  `import 'foo/cjs'` still allows getting two copies (it's less implicit though).

* How would imports/exports merging work when things get hoisted.

* Call out how "browser vs. node" implementation may look like.

* Ecosystem pressure to allow complex resolution.

* The specifier itself can only be mapped using the `main` field to prevent collision.
* `exports` changes the meaning of `main`:
  With `exports`, the absence of `main` means "no mapping of bare bare specifier",
  without `exports`, `main` implicitly defaults to `index`.
* What about `"main": "./index.js"` with `"type": "module"`.
  Would that mean `require('x')` would load the same file as a script
  (and supposedly fail unless it's a global side effect file)?
  Would it fail because `require._extensions` is adjusted based on `type`?
  Myles says he doesn't care about this case. It's mostly about the practical
  impact, not about 100% theoretical correctness.

* Can we still provide shorthands without expansive changes to the CJS loader?
  If we want to support `@quinnjs/cli/runner`,
  the CJS loader would be made to be `package.json` boundary aware because it
  would need to check the package directory for a potential `exports` spec.
  This would be a clear departure from how the resolution algorithm works today
  (`package.json` and `node_modules` have no connection from the CJS perspective).

* What needs to change in the CJS resolution algorithm?

* What are the risks of future browser compat / spec compliance?
  How can we reduce those risks?

* `PACKAGE_EXPORT_RESOLVE(packageName, packageSubpath, parentURL)`
  - Replaces `PACKAGE_MAIN_RESOLVE`
  - If packageSubpath is `""`, it uses the `main` field.
  - Otherwise, it uses the `exports` field (or appends if there is none)

* Using `~/public-entry` may allow us to cleanly support both hoisted and internal
  mappings. Alternatively, we could split this proposal into `imports` and `exports`.
  The `~` syntax could also be used to express "import from myself"

  as a shorthand for the package name.
  It would only be valid for its own scope.
  This should be treated as a future enhancement and maybe should also be opt-in
  because it requires duplicating all import map entries.

* Since it would apply to CJS as well, it would be less confusing to make lock-down
  an independent feature that maybe should not be included in the first version.
  The default value for `exports` would be `{ "./", "./" }`, effectively preserving
  the current behavior of `require`.

Scrap book:

```js
{
  "name": "gofer",
  "main": "lib/gofer.js",
  // General intuition: This is what this package contributes to the import map.
  // It currently includes both information of how it would expect to be used from the outside
  // and how the import map for its internal files ("import map scope") would look like.
  // A possible improvement would be to split these into two separate entries:
  // `exports` (hoisted) and `imports` (internal).
  "exports": { // default "exports": { "/": "/" },
    // These entries will be included in the scope of the user/referrer of this package.
    "/esm": "./lib/gofer.mjs",
    // "my-pkg/esm.mjs" containing "export * from './lib/gofer.mjs'" <- this is the alternative
  },
  "imports": {
    // Entries that start with a "." refer to URLs relative to the package root.
    // When importing the given URL from inside of this scope, it will be rewritten.
    // This process is not recursive (see: import maps).
    "./lib/fetch.js": "./lib/fetch.node.js",
    // This isn't restricted to relative URLs. The same would also work for absolute URLs.
    // Ignore the fallback syntax if it's confusing, this is modelled after:
    //
https://github.com/WICG/import-maps#for-built-in-modules-in-browsers-without-import-maps
    "https://cdnjs.cloudflare.com/ajax/libs/fetch/3.0.0/fetch.min.js": [
      "std:fetch",
      "node:fetch",
      "https://cdnjs.cloudflare.com/ajax/libs/fetch/3.0.0/fetch.min.js",
    ],

    "./lib/request.js": [
      "./lib/request.node.js",
```

```
    "./lib/request.browser.js",
  ],
  "./build/Release/native.node": [
    "./build/Release/native.windows.node",
    "./build/Release/native.osx.node",
    "./build/Release/native.linux.node",
  ],

  // Since everything that doesn't start with `~` is just included in the import map scope,
  // we can also include arbitrary entries. Somebody wants to remap `fs`? They *could*.
  "fs": "https://unpkg.com/graceful-fs@4.1.15/graceful-fs.js",
},

// Now for the crazy bit: We *could*, in theory, replace #dependencies as well.
// Downside: One loader wouldn't be able to resolve the same semver range
// in different ways. Multiple copies still work.
"imports": {
  "express": "https://unpkg.com/express@4.x",
  // Potential sugar:
  // Though it would be interesting how it would affect hoisting of import maps in express.
  // E.g. would that be part of installing it / generating the lock file?
  // Simplest case would be to have 1-2 entries and just use the slash expansion:
  "express/": "https://unpkg.com/express@4.x/",
  // But that isn't actually the same thing, it uses redirects instead of rewrites before fetching.
  // It *could* also mean that the file at https://unpkg.com/express@4.x/ contains something
  // along the lines of:
  //    export * from 'https://unpkg.com/express@4.x/';
  // Although that wouldn't properly handle default exports..?
  // https://github.com/guybedford/proposal-export-star-default never happened. :(

  // One possible way out of this: Use a custom protocol that must be replaced before real use,
  // e.g. when generating the import map.
  // This means the package would (de-facto) not work with the default loader
  // unless it has special support for this.
  // The following may load metadata from the given URL, determine
  // the exports, and generate multiple entries in the import map accordingly.
  "figgy-pudding": "js-bundle+https://legacy@registry.entropic.dev/figgy-pudding",
  // While the above isn't usable in a browser as-is (yet?), the final composed
  // import map would be compatible. It would unlikely to be anything but sugar
  // for package managers and bundlers (and/or node).

  // The easiest solution may be to keep using a separate field (or file) for dependency lists.
},
```

```
}
```