





 [WICG / import-maps](#) Public

How to control the behavior of JavaScript imports

wicg.github.io/import-maps/ View license 2k stars  49 forks Star Watch ▾ Code Issues 17 Pull requests 4 Actions Security Insights main ▾

...



domenic Explain that manual sanitizers need to be careful ...

on 14 Oct 2021

 154

[View code](#) README.md

Import maps

Or, how to control the behavior of JavaScript imports

Table of contents

- [The basic idea](#)
- [Background](#)
- [The import map](#)
 - [Specifier remapping examples](#)
 - [Bare specifiers for JavaScript modules](#)
 - ["Packages" via trailing slashes](#)
 - [Extension-less imports](#)
 - [General URL-like specifier remapping](#)
 - [Mapping away hashes in script filenames](#)
 - [Remapping doesn't work for <script>](#)

- [Scoping examples](#)
 - [Multiple versions of the same module](#)
 - [Scope inheritance](#)
- [Import map processing](#)
 - [Installation](#)
 - [Dynamic import map example](#)
 - [Scope](#)
 - [Interaction with speculative parsing/fetching](#)
- [Feature detection](#)
- [Alternatives considered](#)
 - [The Node.js module resolution algorithm](#)
 - [A programmable resolution hook](#)
 - [Ahead-of-time rewriting](#)
 - [Service workers](#)
 - [A convention-based flat mapping](#)
- [Adjacent concepts](#)
 - [Supplying out-of-band metadata for each module](#)
- [Further work](#)
 - [Multiple import map support](#)
 - [Programmatic API](#)
 - `import.meta.resolve()`
- [Community polyfills and tooling](#)
- [Acknowledgments](#)

The basic idea

This proposal allows control over what URLs get fetched by JavaScript `import` statements and `import()` expressions. This allows "bare import specifiers", such as `import moment` from `"moment"`, to work.

The mechanism for doing this is via an *import map* which can be used to control the resolution of module specifiers generally. As an introductory example, consider the code

```
import moment from "moment";  
import { partition } from "lodash";
```

Today, this throws, as such bare specifiers [are explicitly reserved](#). By supplying the browser with the following import map

```
<script type="importmap">
{
  "imports": {
    "moment": "/node_modules/moment/src/moment.js",
    "lodash": "/node_modules/lodash-es/lodash.js"
  }
}
</script>
```

the above would act as if you had written

```
import moment from "/node_modules/moment/src/moment.js";
import { partition } from "/node_modules/lodash-es/lodash.js";
```

For more on the new "importmap" value for <script>'s type="" attribute, see the [installation section](#). For now, we'll concentrate on the semantics of the mapping, deferring the installation discussion.

Background

Web developers with experience with pre-ES2015 module systems, such as CommonJS (either in Node or bundled using webpack/browserify for the browser), are used to being able to import modules using a simple syntax:

```
const $ = require("jquery");
const { pluck } = require("lodash");
```

Translated into the language of JavaScript's built-in module system, these would be

```
import $ from "jquery";
import { pluck } from "lodash";
```

In such systems, these bare import specifiers of "jquery" or "lodash" are mapped to full filenames or URLs. In more detail, these specifiers represent *packages*, usually distributed on [npm](#); by only specifying the name of the package, they are implicitly requesting the main module of that package.

The main benefit of this system is that it allows easy coordination across the ecosystem. Anyone can write a module and include an import statement using a package's well-known name, and let the Node.js runtime or their build-time tooling take care of translating it into an actual file on disk (including figuring out versioning considerations).

Today, many web developers are even using JavaScript's native module syntax, but combining it with bare import specifiers, thus making their code unable to run on the web without per-application, ahead-of-time modification. We'd like to solve that, and bring these benefits to the web.

The import map

We explain the features of the import map via a series of examples.

Specifier remapping examples

Bare specifiers for JavaScript modules

As mentioned in the introduction,

```
{
  "imports": {
    "moment": "/node_modules/moment/src/moment.js",
    "lodash": "/node_modules/lodash-es/lodash.js"
  }
}
```

gives bare import specifier support in JavaScript code:

```
import moment from "moment";
import("lodash").then(_ => ...);
```

Note that the right-hand side of the mapping (known as the "address") must start with `/`, `../`, or `./`, or be parseable as an absolute URL, to identify a URL. In the case of relative-URL-like addresses, they are resolved relative to the import map's base URL, i.e. the base URL of the page for inline import maps, and the URL of the import map resource for external import maps.

In particular, "bare" relative URLs like `node_modules/moment/src/moment.js` will not work in these positions, for now. This is done as a conservative default, as in the future we may want to allow [multiple import maps](#), which might change the meaning of the right-hand side in ways that especially affect these bare cases.

"Packages" via trailing slashes

It's common in the JavaScript ecosystem to have a package (in the sense of [npm](#)) contain multiple modules, or other files. For such cases, we want to map a prefix in the module specifier space, onto another prefix in the fetchable-URL space.

Import maps do this by giving special meaning to specifier keys that end with a trailing slash. Thus, a map like

```
{
  "imports": {
    "moment": "/node_modules/moment/src/moment.js",
    "moment/": "/node_modules/moment/src/",
    "lodash": "/node_modules/lodash-es/lodash.js",
    "lodash/": "/node_modules/lodash-es/"
  }
}
```

would allow not only importing the main modules like

```
import moment from "moment";
import _ from "lodash";
```

but also non-main modules, e.g.

```
import localeData from "moment/locale/zh-cn.js";
import fp from "lodash/fp.js";
```

Extension-less imports

It is also common in the Node.js ecosystem to import files without including the extension. [We do not have the luxury](#) of trying multiple file extensions until we find a good match. However, we can emulate something similar by using an import map. For example,

```
{
  "imports": {
    "lodash": "/node_modules/lodash-es/lodash.js",
```

```
"lodash/": "/node_modules/lodash-es/",  
"lodash/fp": "/node_modules/lodash-es/fp.js",  
}  
}
```

would allow not only `import fp from "lodash/fp.js"`, but also allow `import fp from "lodash/fp"`.

Although this example shows how it is *possible* to allow extension-less imports with import maps, it's not necessarily *desirable*. Doing so bloats the import map, and makes the package's interface less simple—both for humans and for tooling.

This bloat is especially problematic if you need to allow extension-less imports within a package. In that case you will need an import map entry for every file in the package, not just the top-level entry points. For example, to allow `import "./fp"` from within the `/node_modules/lodash-es/lodash.js` file, you would need an import entry mapping `/node_modules/lodash-es/fp` to `/node_modules/lodash-es/fp.js`. Now imagine repeating this for every file referenced without an extension.

As such, we recommend caution when employing patterns like this in your import maps, or writing modules. It will be simpler for the ecosystem if we don't rely on import maps to patch up file-extension related mismatches.

General URL-like specifier remapping

As part of allowing general remapping of specifiers, import maps specifically allow remapping of URL-like specifiers, such as `"https://example.com/foo.mjs"` or `"./bar.mjs"`. A practical use for this is [mapping away hashes](#), but here we demonstrate some basic ones to communicate the concept:

```
{  
  "imports": {  
    "https://www.unpkg.com/vue/dist/vue.runtime.esm.js": "/node_modules/vue/dist/vue"  
  }  
}
```

This remapping ensures that any imports of the unpkg.com version of Vue (at least at that URL) instead grab the one from the local server.

```
{  
  "imports": {  
    "/app/helpers.mjs": "/app/helpers/index.mjs"  
  }  
}
```

```
}  
}
```

This remapping ensures that any URL-like imports that resolve to `/app/helpers.mjs`, including e.g. an `import "../helpers.mjs"` from files inside `/app/`, or an `import "../helpers.mjs"` from files inside `/app/models`, will instead resolve to `/app/helpers/index.mjs`. This is probably not a good idea; instead of creating an indirection which obfuscates your code, you should instead just update your source files to import the correct files. But, it is a useful example for demonstrating the capabilities of import maps.

Such remapping can also be done on a prefix-matched basis, by ending the specifier key with a trailing slash:

```
{  
  "imports": {  
    "https://www.unpkg.com/vue/": "/node_modules/vue/"  
  }  
}
```

This version ensures that import statements for specifiers that start with the substring `"https://www.unpkg.com/vue/"` will be mapped to the corresponding URL underneath `/node_modules/vue/`.

In general, the point is that the remapping works the same for URL-like imports as for bare imports. Our previous examples changed the resolution of specifiers like `"lodash"`, and thus changed the meaning of `import "lodash"`. Here we're changing the resolution of specifiers like `"/app/helpers.mjs"`, and thus changing the meaning of `import "/app/helpers.mjs"`.

Note that this trailing-slash variant of URL-like specifier mapping only works if the URL-like specifier has a [special scheme](#): e.g., a mapping of `"data:text/": "/foo"` will not impact the meaning of `import "data:text/javascript,console.log('test')"`, but instead will only impact `import "data:text/"`.

Mapping away hashes in script filenames

Script files are often given a unique hash in their filename, to improve cachability. See [this general discussion of the technique](#), or [this more JavaScript- and webpack-focused discussion](#).

With module graphs, this technique can be problematic:

- Consider a simple module graph, with `app.mjs` depending on `dep.mjs` which depends on `sub-dep.mjs`. Normally, if you upgrade or change `sub-dep.mjs`, `app.mjs` and `dep.mjs` can remain cached, requiring only transferring the new `sub-dep.mjs` over the network.
- Now consider the same module graph, using hashed filenames for production. There we have our build process generating `app-8e0d62a03.mjs`, `dep-16f9d819a.mjs`, and `sub-dep-7be2aa47f.mjs` from the original three files.

If we upgrade or change `sub-dep.mjs`, our build process will re-generate a new filename for the production version, say `sub-dep-5f47101dc.mjs`. But this means we need to change the `import` statement in the production version of `dep.mjs`. That changes its contents, which means the production version of `dep.mjs` itself needs a new filename. But then this means we need to update the `import` statement in the production version of `app.mjs` ...

That is, with module graphs and `import` statements containing hashed-filename script files, updates to any part of the graph become viral to all its dependencies, losing all the cachability benefits.

Import maps provide a way out of this dilemma, by decoupling the module specifiers that appear in `import` statements from the URLs on the server. For example, our site could start out with an import map like

```
{
  "imports": {
    "/js/app.mjs": "/js/app-8e0d62a03.mjs",
    "/js/dep.mjs": "/js/dep-16f9d819a.mjs",
    "/js/sub-dep.mjs": "/js/sub-dep-7be2aa47f.mjs"
  }
}
```

and with import statements that are of the form `import "./sub-dep.mjs"` instead of `import "/js/sub-dep-7be2aa47f.mjs"`. Now, if we change `sub-dep.mjs`, we simply update our import map:

```
{
  "imports": {
    "/js/app.mjs": "/js/app-8e0d62a03.mjs",
    "/js/dep.mjs": "/js/dep-16f9d819a.mjs",
    "/js/sub-dep.mjs": "/js/sub-dep-5f47101dc.mjs"
  }
}
```


and leave the `import "./sub-dep.mjs"` statement alone. This means the contents of `dep.mjs` don't change, and so it stays cached; the same for `app.mjs`.

Remapping doesn't work for `<script>`

An important note about using import maps to change the meaning of import specifiers is that it does not change the meaning of raw URLs, such as those that appear in `<script src="">` or `<link rel="modulepreload">`. That is, given the above example, while

```
import "./app.mjs";
```

would be correctly remapping to its hashed version in import-map-supporting browsers,

```
<script type="module" src="./app.mjs"></script>
```

would not: in all classes of browsers, it would attempt to fetch `app.mjs` directly, resulting in a 404. What *would* work, in import-map-supporting browsers, would be

```
<script type="module">import "./app.mjs"</script>
```

Scoping examples

Multiple versions of the same module

It is often the case that you want to use the same import specifier to refer to multiple versions of a single library, depending on who is importing them. This encapsulates the versions of each dependency in use, and avoids [dependency hell](#) ([longer blog post](#)).

We support this use case in import maps by allowing you to change the meaning of a specifier within a given *scope*:

```
{
  "imports": {
    "querystringify": "/node_modules/querystringify/index.js"
  },
  "scopes": {
    "/node_modules/socksjs-client/": {
      "querystringify": "/node_modules/socksjs-client/querystringify/index.js"
    }
  }
}
```

(This is example is one of several [in-the-wild examples](#) of multiple versions per application provided by @zkat. Thanks, @zkat!)

With this mapping, inside any modules whose URLs start with `/node_modules/socksjs-client/`, the `"querystringify"` specifier will refer to `/node_modules/socksjs-client/querystringify/index.js`. Whereas otherwise, the top-level mapping will ensure that `"querystringify"` refers to `/node_modules/querystringify/index.js`.

Note that being in a scope does not change how an address is resolved; the import map's base URL is still used, instead of e.g. the scope URL prefix.

Scope inheritance

Scopes "inherit" from each other in an intentionally-simple manner, merging but overriding as they go. For example, the following import map:

```
{
  "imports": {
    "a": "/a-1.mjs",
    "b": "/b-1.mjs",
    "c": "/c-1.mjs"
  },
  "scopes": {
    "/scope2/": {
      "a": "/a-2.mjs"
    },
    "/scope2/scope3/": {
      "b": "/b-3.mjs"
    }
  }
}
```

would give the following resolutions:

Specifier	Referrer	Resulting URL
a	/scope1/foo.mjs	/a-1.mjs
b	/scope1/foo.mjs	/b-1.mjs
c	/scope1/foo.mjs	/c-1.mjs
a	/scope2/foo.mjs	/a-2.mjs
b	/scope2/foo.mjs	/b-1.mjs

Specifier	Referrer	Resulting URL
c	/scope2/foo.mjs	/c-1.mjs
a	/scope2/scope3/foo.mjs	/a-2.mjs
b	/scope2/scope3/foo.mjs	/b-3.mjs
c	/scope2/scope3/foo.mjs	/c-1.mjs

Import map processing

Installation

You can install an import map for your application using a `<script>` element, either inline or with a `src=""` attribute:

```
<script type="importmap">
{
  "imports": { ... },
  "scopes": { ... }
}
</script>
```

```
<script type="importmap" src="import-map.importmap"></script>
```

When the `src=""` attribute is used, the resulting HTTP response must have the MIME type `application/importmap+json`. (Why not reuse `application/json`? Doing so could [enable CSP bypasses](#).) Like module scripts, the request is made with CORS enabled, and the response is always interpreted as UTF-8.

Because they affect all imports, any import maps must be present and successfully fetched before any module resolution is done. This means that module graph fetching is blocked on import map fetching.

This means that the inline form of import maps is *strongly recommended* for best performance. This is similar to the best practice of [inlining critical CSS](#); both types of resources block your application from doing important work until they're processed, so introducing a second network round-trip (or even disk-cache round trip) is a bad idea. If your heart is set on using external import maps, you can attempt to mitigate this round-trip penalty with technologies like HTTP/2 Push or [bundled HTTP exchanges](#).

As another consequence of how import maps affect all imports, attempting to add a new `<script type="importmap">` after any module graph fetching has started is an error. The import map will be ignored, and the `<script>` element will fire an `error` event.

For now, only one `<script type="importmap">` is allowed on the page. We plan to extend this in the future, once we figure out the correct semantics for combining multiple import maps. See discussion in [#14](#), [#137](#), and [#167](#).

What do we do in workers? Probably `new Worker(someURL, { type: "module", importMap: ... })`? Or should you set it from inside the worker? Should dedicated workers use their controlling document's map, either by default or always? Discuss in [#2](#).

Dynamic import map example

The above rules mean that you *can* dynamically generate import maps, as long as you do so before performing any imports. For example:

```
<script>
const im = document.createElement('script');
im.type = 'importmap';
im.textContent = JSON.stringify({
  imports: {
    'my-library': Math.random() > 0.5 ? '/my-awesome-library.mjs' : '/my-rad-library
  }
});
document.currentScript.after(im);
</script>

<script type="module">
import 'my-library'; // will fetch the randomly-chosen URL
</script>
```

A more realistic example might use this capability to assemble the import map based on feature detection:

```
<script>
const importMap = {
  imports: {
    moment: '/moment.mjs',
    lodash: someFeatureDetection() ?
      '/lodash.mjs' :
      '/lodash-legacy-browsers.mjs'
  }
};
```

```
const im = document.createElement('script');
im.type = 'importmap';
im.textContent = JSON.stringify(importMap);
document.currentScript.after(im);
</script>

<script type="module">
import _ from "lodash"; // will fetch the right URL for this browser
</script>
```

Note that (like other `<script>` elements) modifying the contents of a `<script type="importmap">` after it's already inserted in the document will not work. This is why we wrote the above example by assembling the contents of the import map before creating and inserting the `<script type="importmap">`.

Scope

Import maps are an application-level thing, somewhat like service workers. (More formally, they would be per-module map, and thus per-realm.) They are not meant to be composed, but instead produced by a human or tool with a holistic view of your web application. For example, it would not make sense for a library to include an import map; libraries can simply reference modules by specifier, and let the application decide what URLs those specifiers map to.

This, in addition to general simplicity, is in part what motivates the above restrictions on `<script type="importmap">`.

Since an application's import map changes the resolution algorithm for every module in the module map, they are not impacted by whether a module's source text was originally from a cross-origin URL. If you load a module from a CDN that uses bare import specifiers, you'll need to know ahead of time what bare import specifiers that module adds to your app, and include them in your application's import map. (That is, you need to know what all of your application's transitive dependencies are.) It's important that control of which URLs are used for each package stay with the application author, so they can holistically manage versioning and sharing of modules.

Interaction with speculative parsing/fetching

Most browsers have a speculative HTML parser which tries to discover resources declared in HTML markup while the HTML parser is waiting for blocking scripts to be fetched and executed. This is not yet specified, although there are ongoing efforts to do so in [whatwg/html#5959](https://whatwg.org/html#5959). This section discusses some of the potential interactions to be aware of.

First, note that although to our knowledge no browsers do so currently, it would be possible for a speculative parser to fetch `https://example.com/foo.mjs` in the following example, while it waits for the blocking script `https://example.com/blocking-1.js` :

```
<!DOCTYPE html>
<!-- This file is https://example.com/ -->
<script src="blocking-1.js"></script>
<script type="module">
import "./foo.mjs";
</script>
```

Similarly, a browser could speculatively fetch `https://example.com/foo.mjs` and `https://example.com/bar.mjs` in the following example, by parsing the import map as part of the speculative parsing process:

```
<!DOCTYPE html>
<!-- This file is https://example.com/ -->
<script src="blocking-2.js"></script>
<script type="importmap">
{
  "imports": {
    "foo": "./foo.mjs",
    "https://other.example.com/bar.mjs": "./bar.mjs"
  }
}
</script>
<script type="module">
import "foo";
import "https://other.example.com/bar.mjs";
</script>
```

One interaction to notice here is that browsers which do speculatively parse inline JS modules, but do not support import maps, would probably speculate incorrectly for this example: they might speculatively fetch `https://other.example.com/bar.mjs` , instead of the `https://example.com/bar.mjs` it is mapped to.

More generally, import map-based speculations can be subject to the same sort of mistakes as other speculations. For example, if the contents of `blocking-1.js` were

```
const el = document.createElement("base");
el.href = "/subdirectory/";
document.currentScript.after(el);
```

then the speculative fetch of `https://example.com/foo.mjs` in the no-import map example would be wasted, as by the time came to perform the actual evaluation of the module, we'd re-compute the relative specifier `"./foo.mjs"` and realize that what's actually requested is `https://example.com/subdirectory/foo.mjs`.

Similarly for the import map case, if the contents of `blocking-2.js` were

```
document.write(`<script type="importmap">
{
  "imports": {
    "foo": "./other-foo.mjs",
    "https://other.example.com/bar.mjs": "./other-bar.mjs"
  }
}
</script>`);
```

then the speculative fetches of `https://example.com/foo.mjs` and `https://example.com/bar.mjs` would be wasted, as the newly-written import map would be in effect instead of the one that was seen inline in the HTML.

Feature detection

If the browser supports `HTMLScriptElement`'s `supports(type)` method, `HTMLScriptElement.supports('importmap')` must return true.

```
if (HTMLScriptElement.supports && HTMLScriptElement.supports('importmap')) {
  console.log('Your browser supports import maps.');
```

```
}
```

Alternatives considered

The Node.js module resolution algorithm

Unlike in Node.js, in the browser we don't have the luxury of a reasonably-fast file system that we can crawl looking for modules. Thus, we cannot implement the [Node module resolution algorithm](#) directly; it would require performing multiple server round-trips for every `import` statement, wasting bandwidth and time as we continue to get 404s. We need to ensure that every `import` statement causes only one HTTP request; this necessitates some measure of precomputation.

A programmable resolution hook

Some have suggested customizing the browser's module resolution algorithm using a JavaScript hook to interpret each module specifier.

Unfortunately, this is fatal to performance; jumping into and back out of JavaScript for every edge of a module graph drastically slows down application startup. (Typical web applications have on the order of thousands of modules, with 3-4× that many import statements.) You can imagine various mitigations, such as restricting the calls to only bare import specifiers or requiring that the hook take batches of specifiers and return batches of URLs, but in the end nothing beats precomputation.

Another issue with this is that it's hard to imagine a useful mapping algorithm a web developer could write, even if they were given this hook. Node.js has one, but it is based on repeatedly crawling the filesystem and checking if files exist; we as discussed above, that's infeasible on the web. The only situation in which a general algorithm would be feasible is if (a) you never needed per-subgraph customization, i.e. only one version of every module existed in your application; (b) tooling managed to arrange your modules ahead of time in some uniform, predictable fashion, so that e.g. the algorithm becomes `"return /js/${specifier}.js "`. But if we're in this world anyway, a declarative solution would be simpler.

Ahead-of-time rewriting

One solution in use today (e.g. in the [unpkg](#) CDN via [babel-plugin-unpkg](#)) is to rewrite all bare import specifiers to their appropriate absolute URLs ahead of time, using build tooling. This could also be done at install time, so that when you install a package using npm, it automatically rewrites the package's contents to use absolute or relative URLs instead of bare import specifiers.

The problem with this approach is that it does not work with dynamic `import()`, as it's impossible to statically analyze the strings passed to that function. You could inject a fixup that, e.g., changes every instance of `import(x)` into `import(specifierToURL(x, import.meta.url))`, where `specifierToURL` is another function generated by the build tool. But in the end this is a fairly leaky abstraction, and the `specifierToURL` function largely duplicates the work of this proposal anyway.

Service workers

At first glance, service workers seem like the right place to do this sort of resource translation. We've talked in the past about finding some way to pass the specifier along with a service worker's fetch event, thus allowing it to give back an appropriate `Response`.

However, *service workers are not available on first load*. Thus, they can't really be a part of the critical infrastructure used to load modules. They can only be used as a progressive enhancement on top of fetches that will otherwise generally work.

A convention-based flat mapping

If you have a simple applications with no need for scoped dependency resolution, and have a package installation tool which is comfortable rewriting paths on disk inside the package (unlike current versions of npm), you could get away with a much simpler mapping. For example, if your installation tool created a flat listing of the form

```
node_modules_flattened/  
  lodash/  
    index.js  
    core.js  
    fp.js  
  moment/  
    index.js  
  html-to-dom/  
    index.js
```

then the only information you need is

- A base URL (in our app, `/node_modules_flattened/`)
- The main module filename used (in our app, `index.js`)

You could imagine a module import configuration format that only specified these things, or even only some subset (if we baked in assumptions for the others).

This idea does not work for more complex applications which need scoped resolution, so we believe the full import map proposal is necessary. But it remains attractive for simple applications, and we wonder if there's a way to make the proposal also have an easy-mode that does not require listing all modules, but instead relies on conventions and tools to ensure minimal mapping is needed. Discuss in [#7](#).

Adjacent concepts

Supplying out-of-band metadata for each module

Several times now it's come up that people desire to supply metadata for each module; for example, [integrity metadata](#), or fetching options. Although some have proposed doing this with an import statement, [careful consideration of the options](#) leads to preferring an out-of-band manifest file.

The import map *could* be that manifest file. However, it may not be the best fit, for a few reasons:

- As currently envisioned, most modules in an application would not have entries in the import map. The main use case is for modules you need to refer to by bare specifiers, or modules where you need to do something tricky like polyfilling or virtualizing. If we envisioned every module being in the map, we would not include convenience features like packages-via-trailing-slashes.
- All proposed metadata so far is applicable to any sort of resource, not just JavaScript modules. A solution should probably work at a more general level.

Further work

Multiple import map support

It is natural for multiple `<script type="importmap">`s to appear on a page, just as multiple `<script>`s of other types can. We would like to enable this in the future.

The biggest challenge here is deciding how the multiple import maps compose. That is, given two import maps which both remap the same URL, or two scope definitions which cover the same URL prefix space, what should the affect on the page be? The current leading candidate is [cascading resolution](#), which recasts import maps from being import specifier → URL mappings, to instead be a cascading series of import specifier → import specifier mappings, eventually bottoming out in a "fetchable import specifier" (essentially a URL).

See [these open issues](#) for more discussion.

Programmatic API

Some use cases desire a way of reading or manipulating a realm's import map from script, instead of via inserting declarative `<script type="importmap">` elements. Consider it an "import map object model", similar to the CSS object model that allows one to manipulate the page's usually-declarative CSS rules.

The challenges here are around how to reconcile the declarative import maps with any programmatic changes, as well as when in the page's lifecycle such an API can operate. In general, the simpler designs are less powerful and may meet fewer use cases.

See [these open issues](#) for more discussion and use cases where a programmatic API could help.

`import.meta.resolve()`

The proposed `import.meta.resolve(specifier)` function allows module scripts to resolve import specifiers to URLs at any time. See [whatwg/html#5572](https://whatwg.org/html#5572) for more. This is related to import maps since it allows you to resolve "package-relative" resources, e.g.

```
const url = import.meta.resolve("somepackage/resource.json");
```

would give you the appropriately-mapped location of `resource.json` within the `somepackage/` namespace controlled by the page's import map.

Community polyfills and tooling

Several members of the community have been working on polyfills and tooling related to import maps. Here are the ones we know about:

- [@import-maps/resolve](#) resolves a specifier relative to an import map.
- [@jsenv/importmap-node-module](#) generates an import map from your `package.json` and `node_modules/` directories.
- [@jsenv/importmap-eslint-resolver](#) enables import map resolution in [ESLint](#)
- [@node-loader/import-maps](#) is a [Node.js loader](#) for using import maps in Node.js.
- [@web/dev-server-import-maps](#) allows using import maps during development and testing with [@web/dev-server](#) and [@web/test-runner](#).
- [Deno](#) is a JavaScript/TypeScript runtime with [built-in support for import maps](#).

- [es-module-shims](#) provides an import maps polyfill for browsers with basic ES modules support.
- [import-map-deployer](#) is designed for updating import map files from CI
- [import-map-overrides](#) allows using import maps to improve development flow by pointing to local versions.
- [importly](#) generates an import map from a `package.json`.
- [SystemJS](#) provides a polyfill-like workflow for using import maps in older browsers with the System module format and `<script type="systemjs-importmap">`.
- [webpack-import-map-plugin](#) generates import maps for the output of [webpack](#), especially useful for the [hashing use case](#).
- [import_map](#) is a Rust crate implementing the specification. Used by Deno.

Feel free to send a pull request with more! Also, you can use [#146](#) in the issue tracker for discussion about this space.

Acknowledgments

This document originated out of a day-long sprint involving [@domenic](#), [@hiroshige-g](#), [@justinfagnani](#), [@MylesBorins](#), and [@nyaxt](#). Since then, [@guybedford](#) has been instrumental in prototyping and driving forward discussion on this proposal.

Thanks also to all of the contributors on the issue tracker for their help in evolving the proposal!

Releases

No releases published

Packages


No packages published

Contributors 21



+ 10 contributors

Environments 1

 github-pages Active

Languages

HTML 67.6% JavaScript 31.1% Makefile 1.3%