Import Maps

Draft Community Group Report, 12 January 2021

This version:
https://wicg.github.io/import-maps/

Editor:
Domenic Denicola (Google) d@domenic.me

Participate:
GitHub WICG/import-maps (new issue, open issues)

Commits:
GitHub spec.bs commits

Abstract

Import maps allow web pages to control the behavior of JavaScript imports.

Status of this document

This specification was published by the Web Platform Incubator Community Group. It is not a W3C Standard nor is it on the W3C Standards Track. Please note that under the W3C Community Contributor License Agreement (CLA) there is a limited opt-out and other conditions apply. Learn more about W3C Community and Business Groups.

Table of Contents

File an issue about the selected text

§ 1. Definitions

A ***resolution result*** is either a URL or null.

A ***specifier map*** is an ordered map from strings to resolution results.

A ***import map*** is a struct with two items:

- ***imports***, a specifier map, and
- ***scopes***, an ordered map of URLs to specifier maps.

An ***empty import map*** is an import map with its imports and scopes both being empty maps.

§ 2. Acquiring import maps

§ 2.1. New members of environment settings objects

File an issue about the selected text

Each environment settings object will get an ***import map*** algorithm, which returns an import map created by the first `<script type="importmap">` element that is encountered (before the cutoff).

A `Document` has an import map ***import map***. It is initially a new empty import map.

In set up a window environment settings object, *settings object*'s import map returns the import map of *window*'s associated `Document`.

A `WorkerGlobalScope` has an import map ***import map***. It is initially a new empty import map.

> ISSUE 1 Specify a way to set `WorkerGlobalScope`'s import map. We might want to inherit parent context's import maps, or provide APIs on `WorkerGlobalScope`, but we are not sure. Currently it is always an empty import map. See #2.

In set up a worker environment settings object, *settings object*'s import map returns *worker global scope*'s import map.

> This infrastructure is very similar to the existing specification for module maps.

A `Document` has a ***pending import map script***, which is a `HTMLScriptElement` or null, initially null.

> This is modified by § 2.3 Prepare a script.

Each `Document` has an ***acquiring import maps*** boolean. It is initially true.

> These two pieces of state are used to achieve the following behavior:
>
> - An import map is accepted if and only if it is added (i.e., its corresponding `<script>` element is added) before the first module load is started, even if the loading of the import map file doesn't finish before the first module load is started.
> - Module loading waits for any import map that has already started loading.

§ 2.2. Script type

To process import maps in the prepare a script algorithm consistently with existing script types (i.e. classic or module), we make the following changes:

- Introduce ***import map parse result***, which is a struct with three items:
  - a ***settings object***, an environment settings object;

- an ***import map***, an import map; and
- an ***error to rethrow***, a JavaScript value representing a parse error when non-null.

- the script's type should be either "`classic`", "`module`", or "`importmap`".
- Rename the script's script to ***the script's result***, which can be either a script or an import map parse result.

The following algorithms are updated accordingly:

- prepare a script: see § 2.3 Prepare a script.
- execute a script block Step 4: add the following case.

"`importmap`"

1. Assert: Never reached.

> Import maps are processed by register an import map instead of execute a script block.

> Because we don't make import map parse result the new subclass of script, other script execution-related specs are left unaffected.

§ 2.3. Prepare a script

Inside the prepare a script algorithm, we make the following changes:

- Insert the following step to prepare a script step 7, under "Determine the script's type as follows:":
  - If the script block's type string is an ASCII case-insensitive match for the string "`importmap`", the script's type is "`importmap`".
- Insert the following step before prepare a script step 24:
  - If the script's type is "`importmap`":
    1. If the element's node document's acquiring import maps is false, then queue a task to fire an event named `error` at the element, and return.
    2. Set the element's node document's acquiring import maps to false.

       > In the future we could skip setting acquiring import maps to false, to allow multiple import maps.

    3. Assert: the element's node document's pending import r

- Insert the following case to prepare a script step 24.6:

  `"importmap"`
  : Fetch an import map given *url*, *settings object*, and *options*.

- Insert the following case to prepare a script step 25.2:

  `"importmap"`

  1. Let *import map parse result* be the result of create an import map parse result, given *source text*, *base URL* and *settings object*.
  2. Set the script's result to *import map parse result*.
  3. The script is ready.

- Insert the following case to prepare a script step 26:

  If the script's type is `"importmap"`
  : Set the element's node document's pending import map script to the element. When the script is ready, run the following steps:

    1. Register an import map given the pending import map script.
    2. Set the pending import map script to null.

    > This will (asynchronously) unblock any wait for import maps algorithm instances.

> This is specified similar to the list of scripts that will execute in order as soon as possible.

> CSPs are applied to inline import maps at Step 13 of prepare a script, and to external import maps in fetch an import map, just like applied to classic/module scripts.

To ***fetch an import map*** given *url*, *settings object*, and *options*, run the following steps. This algorithm asynchronously returns an import map or null.

> This algorithm is specified consistently with fetch a single module script steps 5, 7, 8, 9, 10, and 12.1. Particularly, we enforce CORS to avoid leaking the import map contents that shouldn't be accessed.

1. Let *request* be a new request whose url is *url*, destination is `"script"`, mode is `"cors"`, referrer is `"client"`, and client is *settings object*.

   > Here we use `"script"` as the destination, which means the `script-src-elem` CSP directive applies.

2. Set up the module script request given *request* and *options*.

3. Fetch *request*. Return from this algorithm, and run the remaining steps as part of the fetch's process response for the response *response*.

   > *response* is always CORS-same-origin.

4. If any of the following conditions are met, asynchronously complete this algorithm with null, and abort these steps:

   - *response*'s type is `"error"`
   - *response*'s status is not an ok status
   - The result of extracting a MIME type from *response*'s header list is not `"application/importmap+json"`

     > For more context on MIME type checking, see #105 and #119.

5. Let *source text* be the result of UTF-8 decoding response's body.
6. Asynchronously complete this algorithm with the result of create an import map parse result, given *source text*, *response*'s url, and *settings object*.

§ 2.4. Wait for import maps

To ***wait for import maps*** given *settings object*:

1. If *settings object*'s global object is a `Window` object:

   1. Let *document* be *settings object*'s global object's associated `Document`.
   2. Set *document*'s acquiring import maps to false.
   3. Spin the event loop until *document*'s pending import map script is null.

2. Asynchronously complete this algorithm.

> No actions are specified for `WorkerGlobalScope` because for now there are no mechanisms for adding import maps to `WorkerGlobalScope`.

Insert a call to wait for import maps at the beginning of the following HTML spec concepts.

- fetch an external module script graph
- fetch an import() module script graph
- fetch a modulepreload module script graph
- fetch an inline module script graph

File an issue about the selected text

If/when `WorkerGlobalScope` gets import map support, there will also be some impact on fetch a module worker script graph, but it's not clear exactly what.

§ 2.5. Registering an import map

To ***register an import map*** given an `HTMLScriptElement` *element*:

1. If *element*'s the script's result is null, then fire an event named `error` at *element*, and return.
2. Let *import map parse result* be *element*'s the script's result.
3. Assert: *element*'s the script's type is `"importmap"`.
4. Assert: *import map parse result* is an import map parse result.
5. Let *settings object* be *import map parse result*'s settings object.
6. If *element*'s node document's relevant settings object is not equal to *settings object*, then return.

   This is specified consistently with whatwg/html#2673.

   Currently we don't fire `error` events in this case. If we change the decision at whatwg/html#2673 to fire `error` events, then we should change this step accordingly.
7. If *import map parse result*'s error to rethrow is not null, then:

   1. Report the exception given *import map parse result*'s error to rethrow.

      ISSUE 2 There are no relevant script, because import map parse result isn't a script. This needs to wait for whatwg/html#958 before it is fixable.

   2. Return.
8. Set *element*'s node document's import map to *import map parse result*'s import map.
9. If *element* is from an external file, then fire an event named `load` at *element*.

The timing of register an import map is observable by possible `error` and `load` events, or by the fact that after register an import map an import map `<script>` can be moved to another `Document`. On the other hand, the updated import map is not observable until wait for import maps completes.

§ 3. Parsing import maps

File an issue about the selected text

To ***parse an import map string***, given a string *input* and a URL *baseURL*:

1. Let *parsed* be the result of parsing JSON into Infra values given *input*.
2. If *parsed* is not a map, then throw a `TypeError` indicating that the top-level value needs to be a JSON object.
3. Let *sortedAndNormalizedImports* be an empty map.
4. If *parsed*["`imports`"] exists, then:

    1. If *parsed*["`imports`"] is not a map, then throw a `TypeError` indicating that the "`imports`" top-level key needs to be a JSON object.
    2. Set *sortedAndNormalizedImports* to the result of sorting and normalizing a specifier map given *parsed*["`imports`"] and *baseURL*.

5. Let *sortedAndNormalizedScopes* be an empty map.
6. If *parsed*["`scopes`"] exists, then:

    1. If *parsed*["`scopes`"] is not a map, then throw a `TypeError` indicating that the "`scopes`" top-level key needs to be a JSON object.
    2. Set *sortedAndNormalizedScopes* to the result of sorting and normalizing scopes given *parsed*["`scopes`"] and *baseURL*.

7. If *parsed*'s keys contains any items besides "`imports`" or "`scopes`", report a warning to the console that an invalid top-level key was present in the import map.

    > This can help detect typos. It is not an error, because that would prevent any future extensions from being added backward-compatibly.

8. Return the import map whose imports are *sortedAndNormalizedImports* and whose scopes scopes are *sortedAndNormalizedScopes*.

To ***create an import map parse result***, given a string *input*, a URL *baseURL*, and an environment settings object *settings object*:

1. Let *import map* be the result of parse an import map string given *input* and *baseURL*. If this throws an exception, let *error to rethrow* be the exception. Otherwise, let *error to rethrow* be null.
2. Return an import map parse result with settings object is *settings object*, import map is *import map*, and error to rethrow is *error to rethrow*.

EXAMPLE 1

The import map is a highly normalized structure. For example, given a base URL of `<https://example.com/base/page.html>`, the input

```
{
  "imports": {
    "/app/helper": "node_modules/helper/index.mjs",
    "lodash": "/node_modules/lodash-es/lodash.js"
  }
}
```

will generate an import map with imports of

```
«[
  "https://example.com/app/helper" → <https://example.com/base/node_modules/he
  "lodash" → <https://example.com/node_modules/lodash-es/lodash.js>
]»
```

and (despite nothing being present in the input) an empty map for its scopes.

To ***sort and normalize a specifier map***, given a map *originalMap* and a URL *baseURL*:

1. Let *normalized* be an empty map.
2. For each *specifierKey* → *value* of *originalMap*,
    1. Let *normalizedSpecifierKey* be the result of normalizing a specifier key given *specifierKey* and *baseURL*.
    2. If *normalizedSpecifierKey* is null, then continue.
    3. If *value* is not a string, then:
        1. Report a warning to the console that addresses need to be strings.
        2. Set *normalized*[*normalizedSpecifierKey*] to null.
        3. Continue.
    4. Let *addressURL* be the result of parsing a URL-like import specifier given *value* and *baseURL*.
    5. If *addressURL* is null, then:
        1. Report a warning to the console that the address was invalid.
        2. Set *normalized*[*normalizedSpecifierKey*] to null.
        3. Continue.

6. If *specifierKey* ends with U+002F (/), and the serialization of *addressURL* does not end with U+002F (/), then:
    1. Report a warning to the console that an invalid address was given for the specifier key *specifierKey*; since *specifierKey* ended in a slash, the address needs to as well.
    2. Set *normalized*[*normalizedSpecifierKey*] to null.
    3. Continue.
7. Set *normalized*[*normalizedSpecifierKey*] to *addressURL*.

3. Return the result of sorting *normalized*, with an entry *a* being less than an entry *b* if *b*'s key is code unit less than *a*'s key.

To ***sort and normalize scopes***, given a map *originalMap* and a URL *baseURL*:

1. Let *normalized* be an empty map.
2. For each *scopePrefix* → *potentialSpecifierMap* of *originalMap*,
    1. If *potentialSpecifierMap* is not a map, then throw a `TypeError` indicating that the value of the scope with prefix *scopePrefix* needs to be a JSON object.
    2. Let *scopePrefixURL* be the result of parsing *scopePrefix* with *baseURL* as the base URL.
    3. If *scopePrefixURL* is failure, then:
        1. Report a warning to the console that the scope prefix URL was not parseable.
        2. Continue.
    4. Let *normalizedScopePrefix* be the serialization of *scopePrefixURL*.
    5. Set *normalized*[*normalizedScopePrefix*] to the result of sorting and normalizing a specifier map given *potentialSpecifierMap* and *baseURL*.
3. Return the result of sorting *normalized*, with an entry *a* being less than an entry *b* if *b*'s key is code unit less than *a*'s key.

> We sort keys/scopes in reverse order, to put `"foo/bar/"` before `"foo/"` so that `"foo/bar/"` has a higher priority than `"foo/"`.

To ***normalize a specifier key***, given a string *specifierKey* and a URL *baseURL*:

1. If *specifierKey* is the empty string, then:
    1. Report a warning to the console that specifier keys cannot be the empty string.
    2. Return null.

2. Let *url* be the result of parsing a URL-like import specifier, given *specifierKey* and *baseURL*.
3. If *url* is not null, then return the serialization of *url*.
4. Return *specifierKey*.

To ***parse a URL-like import specifier***, given a string *specifier* and a URL *baseURL*:

1. If *specifier* starts with "`/`", "`./`", or "`../`", then:

    1. Let *url* be the result of parsing *specifier* with *baseURL* as the base URL.
    2. If *url* is failure, then return null.

        EXAMPLE 2
        One way this could happen is if *specifier* is "`../foo`" and *baseURL* is a `data:` URL.

    3. Return *url*.

    This includes cases where *specifier* starts with "`//`", i.e. scheme-relative URLs. Thus, *url* might end up with a different host than *baseURL*.

2. Let *url* be the result of parsing *specifier* (with no base URL).
3. If *url* is failure, then return null.
4. Return *url*.

§ 4. Resolving module specifiers

During resolving a module specifier, the following algorithms check candidate entries of specifier maps, from most-specific to least-specific scopes (falling back to top-level "`imports`"), and from most-specific to least-specific prefixes. For each candidate, the result is one of the following:

- Successfully resolves a specifier to a URL. This makes the resolve a module specifier algorithm immediately return that URL.
- Throws an error. This makes the resolve a module specifier algorithm rethrow the error, without any further fallbacks.
- Fails to resolve, without an error. In this case the algorithm moves on to the next candidate.

§ 4.1. New "resolve a module specifier"

File an issue about the selected text

HTML already has a resolve a module specifier algorithm. We replace it with the following ***resolve a module specifier*** algorithm, given a script *referringScript* and a JavaScript string *specifier*:

1. Let *settingsObject* be the current settings object.
2. Let *baseURL* be *settingsObject*'s API base URL.
3. If *referringScript* is not null, then:
    1. Set *settingsObject* to *referringScript*'s settings object.
    2. Set *baseURL* to *referringScript*'s base URL.
4. Let *importMap* be *settingsObject*'s import map.
5. Let *baseURLString* be *baseURL*, serialized.
6. Let *asURL* be the result of parsing a URL-like import specifier given *specifier* and *baseURL*.
7. Let *normalizedSpecifier* be the serialization of *asURL*, if *asURL* is non-null; otherwise, *specifier*.
8. For each *scopePrefix* → *scopeImports* of *importMap*'s scopes,
    1. If *scopePrefix* is *baseURLString*, or if *scopePrefix* ends with U+002F (/) and *baseURLString* starts with *scopePrefix*, then:
        1. Let *scopeImportsMatch* be the result of resolving an imports match given *normalizedSpecifier*, *asURL*, and *scopeImports*.
        2. If *scopeImportsMatch* is not null, then return *scopeImportsMatch*.
9. Let *topLevelImportsMatch* be the result of resolving an imports match given *normalizedSpecifier*, *asURL*, and *importMap*'s imports.
10. If *topLevelImportsMatch* is not null, then return *topLevelImportsMatch*.
11. At this point, the specifier was able to be turned in to a URL, but it wasn't remapped to anything by *importMap*.

    If *asURL* is not null, then return *asURL*.
12. Throw a `TypeError` indicating that *specifier* was a bare specifier, but was not remapped to anything by *importMap*.

To ***resolve an imports match***, given a string *normalizedSpecifier*, a URL or null *asURL*, and a specifier map *specifierMap*:

1. For each *specifierKey* → *resolutionResult* of *specifierMap*,
    1. If *specifierKey* is *normalizedSpecifier*, then:

1. If *resolutionResult* is null, then throw a `TypeError` indicating that resolution of *specifierKey* was blocked by a null entry.

   > This will terminate the entire resolve a module specifier algorithm, without any further fallbacks.

2. Assert: *resolutionResult* is a URL.
3. Return *resolutionResult*.

2. If all of the following are true:

   - *specifierKey* ends with U+002F (/),
   - *normalizedSpecifier* starts with *specifierKey*, and
   - either *asURL* is null, or *asURL* is special

   then:

   1. If *resolutionResult* is null, then throw a `TypeError` indicating that resolution of *specifierKey* was blocked by a null entry.

      > This will terminate the entire resolve a module specifier algorithm, without any further fallbacks.

   2. Assert: *resolutionResult* is a URL.
   3. Let *afterPrefix* be the portion of *normalizedSpecifier* after the initial *specifierKey* prefix.
   4. Assert: *resolutionResult*, serialized, ends with "`/`", as enforced during parsing.
   5. Let *url* be the result of parsing *afterPrefix* relative to the base URL *resolutionResult*.
   6. If *url* is failure, then throw a `TypeError` indicating that resolution of *normalizedSpecifier* was blocked since the *afterPrefix* portion could not be URL-parsed relative to the *resolutionResult* mapped to by the *specifierKey* prefix.

      > This will terminate the entire resolve a module specifier algorithm, without any further fallbacks.

   7. Assert: *url* is a URL.
   8. If the serialization of *url* does not start with the serialization of *resolutionResult*, then throw a `TypeError` indicating that resolution of *normalizedSpecifier* was blocked due to it backtracking above its prefix *specifierKey*.

File an issue about the selected text

This will terminate the entire resolve a module specifier algorithm, without any further fallbacks.

9. Return *url*.

2. Return null.

The resolve a module specifier algorithm will fallback to a less specific scope or to `"imports"`, if possible.

§ 4.2. Updates to other algorithms

All call sites of HTML's existing resolve a module specifier will need to be updated to pass the appropriate script, not just its base URL. Some particular interesting cases:

- HostResolveImportedModule and HostImportModuleDynamically no longer need to compute the base URL themselves, as resolve a module specifier now handles that.
- Fetch an import() module script graph will also need to take a script instead of a base URL.

Call sites will also need to be updated to account for resolve a module specifier now throwing exceptions, instead of returning failure. (Previously most call sites just turned failures into `TypeError`s manually, so this is straightforward.)

§ 5. Security and Privacy

§ 5.1. Threat models

§ 5.1.1. Comparison with first-party scripts

Import maps are explicitly designed to be installed by page authors, i.e. those who have the ability to run first-party scripts. (See the explainer's "Scope" section.)

Although it may seem that the ability to change how resources are imported from JavaScript and the capability of rewriting rules are powerful, there is no extra power really granted here, compared with first-party scripts. That is, they only change things which the page author could change already, by manually editing their code to use different URLs.

We do still need to apply the traditional protections against first-party malicious actors, for example:

- CSP to protect against injection vulnerabilities. (See #105 for further discussion.)

- CORS and strict MIME type checking (with a new MIME type, "`application/importmap+json`") for external import maps.

But there is no fundamentally new capability introduced here, that needs new consideration.

§ 5.1.2. Comparison with Service Workers

On one hand, the ability of import maps to change how resources are imported looks similar to the ability of Service Workers to intercept and rewrite fetch requests.

On the other hand, import maps have a much more restricted scope than Service Workers. Import maps are not persistent, and an import map only affects the document that installs the import map via `<script type="importmap">`.

Therefore, the security restrictions applied to Service Workers (beyond those applied to first-party scripts), e.g. the same-origin/secure contexts requirements, are not applied to import maps.

§ 5.1.3. Time/memory complexity

To avoid denial of service attacks, explosive memory usage, and the like, import maps are designed to have reasonably bounded time and memory complexity in the worst cases, and to not be Turing complete.

§ 5.2. A note on import specifiers

The import specifiers that appear in `import` statements and `import()` expressions are not URLs, and should not be thought of as such.

To date, there has been a default mechanism for translating those strings into URLs. And indeed, some of the strings, such as "`https://example.com/foo.mjs`", or "`./bar.mjs`", might look URL-like; for those, the default translation does what you would expect.

But overall, one should not think of `import(x)` as corresponding to `fetch(x)`. Instead, the correspondence is to `fetch(translate(x))`, where the translation algorithm produces the actual URL to be fetched. In this framing, the way to think about import maps is as providing a mechanism for overriding the default mechanism, i.e. customizing the `translate()` function.

This brings some clarity to some common security questions. For example: given an import map which maps the specifier "`https://1.example.com/foo.mjs`" to the URL `<https://2.example.com/bar.mjs>`, should we apply CSP checks to `<https://1.example.com/foo.mjs>` or to `<https://2.example.com/bar.mjs>`? With this

framing we can see that we should apply the checks to the post-translation URL `<https://2.example.com/bar.mjs>` which is actually fetched, and not to the pre-translation `"https://1.example.com/foo.mjs"` module specifier.

This also makes it clear that other URL-keyed features of the platform, such as as-yet-unspecified the `error.stack` property, will use the post-translation URLs, not the pre-translation module specifiers.

§ Index

§ Terms defined by this specification

§ Terms defined by reference

[CONSOLE] defines the following terms:

- report a warning to the console

File an issue about the selected text

[DOM] defines the following terms:

- Document
- fire an event
- node document

[ENCODING] defines the following terms:

- utf-8 decode

[FETCH] defines the following terms:

- body
- client
- destination
- extracting a mime type
- fetch
- header list
- mode
- ok status
- process response
- referrer
- request
- response
- status
- type
- url (for response)

[HTML] defines the following terms:

- HTMLScriptElement
- Window
- WorkerGlobalScope
- api base url
- associated document
- base url
- cors-same-origin
- current settings object
- environment settings object
- execute a script block
- fetch a module worker script graph
- fetch a modulepreload module script graph
- fetch a single module script
- fetch an external module script graph
- fetch an import() module script graph
- fetch an inline module script graph
- fire an event
- from an external file
- global object
- hostimportmoduledynamically
- hostresolveimportedmodule
- list of scripts that will execute in order as soon as possible
- node document
- prepare a script
- queue a task
- relevant settings object
- report the exception
- resolve a module specifier
- script
- set up a window environment settings object
- set up a worker environment settings object
- set up the module script request
- settings object
- spin the event loop
- the script is ready
- the script's sc

the script's type

[INFRA] defines the following terms:

ascii case-insensitive

code unit less than

contain

continue

exist

for each

get the keys

item

javascript string

key

map

ordered map

parse json into infra values

sorting

starts with

string

struct

[URL] defines the following terms:

host

is special

url

url parser

url serializer

[WebIDL] defines the following terms:

TypeError

File an issue about the selected text

§ References

§ Normative References

[CONSOLE]
Dominic Farolino; Robert Kowalski; Terin Stock. Console Standard. Living Standard. URL: https://console.spec.whatwg.org/

[DOM]
Anne van Kesteren. DOM Standard. Living Standard. URL: https://dom.spec.whatwg.org/

[ENCODING]
Anne van Kesteren. Encoding Standard. Living Standard. URL: https://encoding.spec.whatwg.org/

[FETCH]
Anne van Kesteren. Fetch Standard. Living Standard. URL: https://fetch.spec.whatwg.org/

[HTML]
Anne van Kesteren; et al. HTML Standard. Living Standard. URL: https://html.spec.whatwg.org/multipage/

[INFRA]
Anne van Kesteren; Domenic Denicola. Infra Standard. Living Standard. URL: https://infra.spec.whatwg.org/

[URL]
Anne van Kesteren. URL Standard. Living Standard. URL: https://url.spec.whatwg.org/

[WebIDL]
Boris Zbarsky. Web IDL. URL: https://heycam.github.io/webidl/

§ Issues Index

ISSUE 1 Specify a way to set `WorkerGlobalScope`'s import map. We might want to inherit parent context's import maps, or provide APIs on `WorkerGlobalScope`, but we are not sure. Currently it is always an empty import map. See #2. ↵

ISSUE 2 There are no relevant script, because import map parse result isn't a script. This needs to wait for whatwg/html#958 before it is fixable. ↵

File an issue about the selected text