Layered APIs fallback syntax

by domenic@

Background

The original strawperson syntax for layered web APIs was

```
<script type="module"
    stdsrc="std:infinite-list"                      Browser provided script
    src="https://some.cdn.com/infinite-list.js">    Fallback for older browsers
</script>
```

However, this fallback does not work with JavaScript import syntax:

```
<script type="module">
import { get, set, delete } from "std:async-local-storage"; // no fallback!
// ...
</script>
```

This document explores some alternatives.

Possible goals

It's unclear which of these are must-haves versus nice-to-haves.

- Falls back to the polyfill in browsers that don't implement any layered APIs infrastructure.
- Falls back to the polyfill in browsers that don't implement that specific layered API, but do implement the basic layered API infrastructure.
- The fallback works in <script type="module" src="">.
- The fallback works in JavaScript import statements.
- The fallback works in other URL-accepting locations, like <link href="">, fetch(), or CSS url().

There are also aesthetic considerations, e.g. is it OK to require typing the same URL multiple times; you'll see these in more detail as we go through the alternatives.

Potential solutions

These are ordered in "narrative" order, so you can see the ideas evolve in reaction to the previous one. They're not ranked in order of goodness or anything like that.

A. Original proposal

```
<script type="module"
    stdsrc="std:infinite-list"
    src="https://some.cdn.com/infinite-list.js">
</script>

<script type="module">
import { get, set, delete } from "std:async-local-storage";
// ...
</script>
```

- (+) Works in browsers that don't implement any layered APIs infrastructure
- (-) Doesn't provide a fallback for JavaScript imports
- (-) Requires standardizing and implementing attribute pairs for every place a layered API URL might show up, e.g. `<link rel="stylesheet" href="..." stdhref="...">`.

B. Fallback as part of the std: scheme

```
<script type="module"
        src="std:infinite-list|https://some.cdn.com/infinite-list.js">
</script>

<script type="module">
import { get, set, delete } from
       "std:async-local-storage|https://some-cdn.com/async-local-storage.js";
// ...
</script>
```

- (+) Works in all locations that accept a URL, including JavaScript imports and other tags
- (-) Doesn't work in browsers that don't implement any layered API infrastructure
- (-) A little ugly

C. Fallback + nostd=""

```
<script type="module"
        src="std:infinite-list|https://some.cdn.com/infinite-list.js">
</script>
<script type="module"
        nostd
        src="https://some.cdn.com/infinite-list.js">
</script>
```

```
<script type="module">
import { get, set, delete } from
       "std:async-local-storage|https://some-cdn.com/async-local-storage.js";
// ...
</script>
<script type="module" nostd>
import { get, set, delete } from "https://some-cdn.com/async-local-storage.js";
// ...
</script>
```

- (+) Works in browsers that don't implement any layered APIs infrastructure
- (-) Requires declaring the CDN URL twice
- (-) It requires duplicating your code that actually uses the imports.
- (-) Will cause console errors in older browsers when they try and fail to fetch the std: scheme in the first inline script block
- (-) Requires standardizing and implementing the nostd="" attribute or equivalent anywhere we might use layered API URLs

D. A layered API-specific mapping mechanism, inline

```
<script type="module"
        layered="https://some.cdn.com/infinite-list.js std:infinite-list"
        src="https://some.cdn.com/infinite-list.js">
</script>

<script type="module"
        layered="https://some.cdn.com/async-local-storage.js
                 std:async-local-storage">
import { get, set, delete } from "https://some-cdn.com/async-local-storage.js";
// ...
</script>
```

- (+) Works in browsers that don't implement any layered API infrastructure
- (-) Requires declaring the CDN URL twice
- (-) Requires having this mapping mechanism anywhere we might use layered API URLs
- (-) Usage site ends up using the fallback URLs as primary, which feels strange

D.2 Original proposal with module loading tweaks

```
<script type="module"
    stdsrc="std:infinite-list"
```

```
    src="https://some.cdn.com/infinite-list.js">
</script>

<script type="module"
    stdsrc="std:async-local-storage"
    src="https://some.cdn.com/async-local-storage.js">
</script>

<script type="module">
import { get, set, delete } from "std:async-local-storage";
// ...
</script>
```

In this alternative, the async-local-storage <script> modifies the module map so that "std:async-local-storage" points to https://some.cdn.com/async-local-storage.js if no async-local-storage layered API exists.

- (+) Usage site inside JavaScript modules is simple
- (-) Doesn't work in browsers without any layered APIs infrastructure
- (-) Requires ahead-of-time declaration of any module imports that need fallbacks, as <script>s
- (-) Only works for JavaScript modules

E. A general resource-specifier mapping mechanism

```
<link specifier="jquery" href="/js/libs/jquery.js">
<link specifier="infinite-list"
      href="https://cdn.example.com/infinite-list.js"
      stdhref="std:infinite-list">
<link specifier="async-local-storage"
      href="https://cdn.example.com/async-local-storage.js"
      stdhref="std:async-local-storage">

<script type="module" src="infinite-list"></script>
<script type="module">
import $ from "jquery";
import { get, set, delete } from "async-local-storage";
// ...
</script>
```

- (+) Very concise usage-site code
- (+) We need to solve this bare module specifier problem anyway (see the "jquery" example embedded in the above)
- (+) This general mapping can be used anywhere URLs are usable
- (-) Blocking on this larger problem space is not good for layered APIs

- (-) The actual shape of the solution here is very unclear; e.g. the more general version would be imperative (some kind of fetch worklet), instead of the above simple declarative one; that could be a lot of work to implement
- (-) Does not work in browsers that do not implement this new mechanism

F. A layered API-specific mapping mechanism, ahead of time

```
<link rel="layeredapi" href="https://some.cdn.com/infinite-list.js"
                       stdhref="std:infinite-list">
<link rel="layeredapi" href="https://some.cdn.com/async-local-storage.js"
                       stdhref="std:async-local-storage">

<script type="module"
        src="https://some.cdn.com/infinite-list.js">
</script>

<script type="module">
import { get, set, delete } from "https://some-cdn.com/async-local-storage.js";
// ...
</script>
```

- (+) Works in browsers that don't implement any layered API infrastructure
- (+) Works anywhere URLs are accepted
- (-) May step on the toes of whatever we end up doing for bare module specifiers; at the very least we'd have to define their interaction
- (-) Usage site ends up using the fallback URLs as primary, which feels strange

G. Special-case a layered API domain (ideally TLD)

Inspired by Google Cloud IAM gserviceaccount.com

```
<script type="module"
        src="https://mycdn.layeredapi/infinite-list.js">
</script>

<script type="module">
import { get, set, delete } from "https://mycdn.layeredapi/async-local-storage.js";
// ...
</script>
```

- (+) Works in browsers that don't implement any layered API infrastructure.
- (+) Works anywhere URLs are accepted.
- (+) No URL repetition.
- (-) May be a bit confusing if the user doesn’t know about layered APIs at all.

Variations:

1.
    a. Deploy a DNS server for .layeredapi which resolves to CNAME w/o the .layeredapi suffix, so that this allows fallback to arbitrary domains (not just the specific domain mycdn.layeredapi). This would require a specific hosting structure though, so e.g. it'd have to be https://customcdn.com/async-local-storage.js, and disallow cases like https://cdn.rawgit.com/domenic/als/master/async-local-storage.js.
        i. (+) Less latency involved for unsupported browsers. Single DNS resolve.
        ii. (-) If https, this requires the customcdn.com to also sign customcdn.com.layeredapi domain subject in their TLS certs and their frontend servers to serve content to “Host: customcdn.com.layeredapi”
    b. Deploy a HTTPS redirect server for .layeredapi, which given a URL “https://some.cdn.com.layeredapi/a/b/async-local-storage.js”, redirect to “https://some.cdn.com/a/b/async-local-storage.js”
        i. (+) No additional subjects in TLS certs required.
        ii. (-) Full HTTPS request round-trip latency required for unsupported browsers.
        iii. (-) Users should know the target of redirect if they want to apply strict CSP policies?
2. https://layeredapi.anydomain/infinite-list.js: in supported browsers, if the URL's host has the prefix "layeredapi.", load browser's copy of the layered API instead.
3. https://anydomain/.well-known/layeredapi/infinite-list.js: in supported supported browsers, if the URL's path has the prefix "/.well-known/layeredapi/", load the browser's copy of the layered API instead.

H. Use the integrity="" attribute

```
<script type="module"
        src="https://anyurl/infinite-list.js"
        integrity="sha256-abcdef123">
</script>
<script type="module"
        src="https://anyurl/async-local-storage.js"
        integrity="sha256-d3adb33f456">
</script>

<script type="module">
import { get, set, delete } from "https://anyurl/async-local-storage.js";
// ...
</script>
```

If the integrity="" value is known/registered, resolve to the bundled layered API. Because of the module map caching, you can do this with <script type="module"> first and future loads with import will also work.

- (+) Works in browsers that don't implement any layered APIs infrastructure.
- (-) Does not allow us to change the contents of layered APIs over time, at least with the current definition of integrity as a hash. (This is pretty fatal.)
- (-) Requires repetition for the JavaScript import case.