JavaScript Modules: Past, Present and Future

NICOLÒ RIBAUDO

@NicoloRibaudo https://nicr.dev @nicolo-ribaudo

NICOLÒ RIBAUDO

- Working at Igalia on web standards
- TC39 delegate
- Maintaining Babel, the JavaScript compiler

igalia
Open Source Consultancy

in collaboration with Bloomberg

https://nicr.dev

Welcome to your History class!

The original "modules" system

Back in the day, JavaScript was mostly used to add small bits of interactivity to web pages. There were no "modules", and a few external libraries were loaded installing properties directly on the **`window`** object.

```
<script src="https://code.jquery.com/jquery.min.js"></script>
<script>
  $(document).ready(function() {
      $("#hello").css("color", "red");
  });
</script>
```

2009: CommonJS

What Server Side JavaScript needs

Jan 29, 2009 14:00 · 816 words · 4 minute read

Server side JavaScript technology has been around for a *long* time. Netscape offered server side JavaScript in their server software back in 1996, and Helma has existed for

JavaScript needs a **standard way to include other modules** and for those modules to live in discreet namespaces. There are easy ways to do namespaces, but there's no standard programmatic way to load a module (*once!*). This is really important, because server side apps can include a lot of code and will likely mix and match parts that meet those standard interfaces.

~ Kevin Dangoor

https://www.blueskyonmars.com/2009/01/29/what-server-side-javascript-needs/

2009: CommonJS

```
CommonJSImplementation(function (require, exports, module) {
  var chalk = require("chalk");

  exports.error = function (message) {
    console.log(chalk.red(message));
  };
});
```

Import other modules

Expose values from this module

https://nicr.dev

2009: CommonJS

- An effort to standardize a modules system and a set of built-in APIs across multiple server-side environments

CommonJS Modules/1.1

- Only define the "module interface", implementations must bring their own runtime loader
- Implemented by Node.js, Narshall, SproutCore, and many others

https://arstechnica.com/information-technology/2009/12/commonjs-effort-sets-javascript-on-path-for-world-domination/

2009: CommonJS

CommonJS' `require` is synchronous: how to design a module system that works in the browser?

CommonJS Modules/Transport/C and Modules/AsynchronousDefinition

Pre-declare all the dependencies that need to be loaded, and only start evaluating code once everything is ready.

https://wiki.commonjs.org/wiki/Modules/AsynchronousDefinition
https://github.com/amdjs/amdjs-api/wiki/AMD

2010: AMD

Pre-declare all the dependencies that need to be loaded, and only start evaluating code once everything is ready.

```
define("alpha", ["require", "exports", "beta"], function (require, exports, beta) {
  exports.verb = function() {
    return beta.verb();
    //Or:
    return require("beta").verb();
  }
});
```

https://github.com/amdjs/amdjs-api/wiki/AMD

2010: AMD

Asynchronously load dependencies that cannot be statically declared.

```
define(function (require) {
  require(['i18n/' + lang], function (i18n) {
    // modules i18n is now available for use.
  });
});
```

https://github.com/amdjs/amdjs-api/wiki/AMD

2010: AMD

AMD supports *plugins*, to let developers customize how modules are resolved, loaded and executed.

```
define(['text!../templates/start.html'], function (template) {
  //do something with the template text string.
});
```

2015: ECMAScript Modules

```
// math.js
export function sum() {
  let s = 0;
  for (let x of arguments) s =+ x;
  return s;
}

// main.js
import { sum } from "./math.js";
console.log(sum(1, 2, 3)); // 6
```

2015: ECMAScript Modules

- Statically analyzable: runtimes can preload all the necessary dependencies before executing the code

Like AMD!

- Minimal syntactic overhead (no boilerplate)

Like CommonJS!

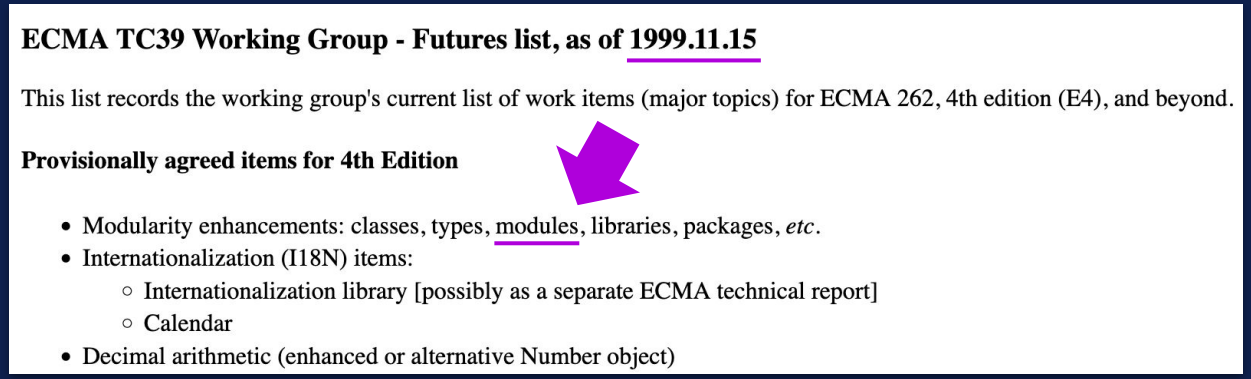
- Support for "named" and "default" exports
  (no overlapping `module.exports` vs `module.exports.name`)

2015: ECMAScript Modules

They are the result of multiple years of development, taking input from different parties and exploring many possible designs and features.

2015: ECMAScript Modules

They are the result of **multiple years** of development, taking input from different parties and exploring many possible designs and features.



ECMA TC39 Working Group - Futures list, as of 1999.11.15

This list records the working group's current list of work items (major topics) for ECMA 262, 4th edition (E4), and beyond.

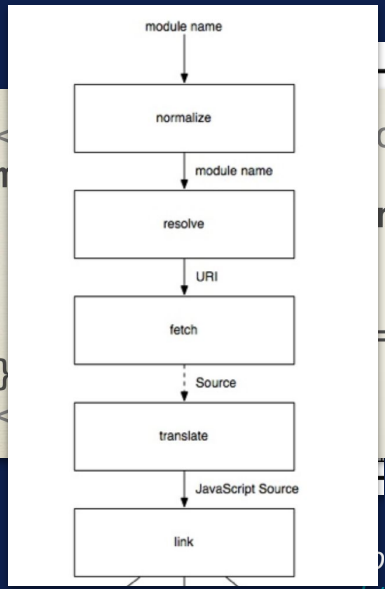
Provisionally agreed items for 4th Edition

- Modularity enhancements: classes, types, modules, libraries, packages, *etc*.
- Internationalization (I18N) items:
  - Internationalization library [possibly as a separate ECMA technical report]
  - Calendar
- Decimal arithmetic (enhanced or alternative Number object)

https://archives.ecma-international.org/1999/TC39WG/991115-futures.htm

2015: ECMAScript Modules

They are the result of multiple years of development, taking input from different parties and exploring **many possible designs and features**.



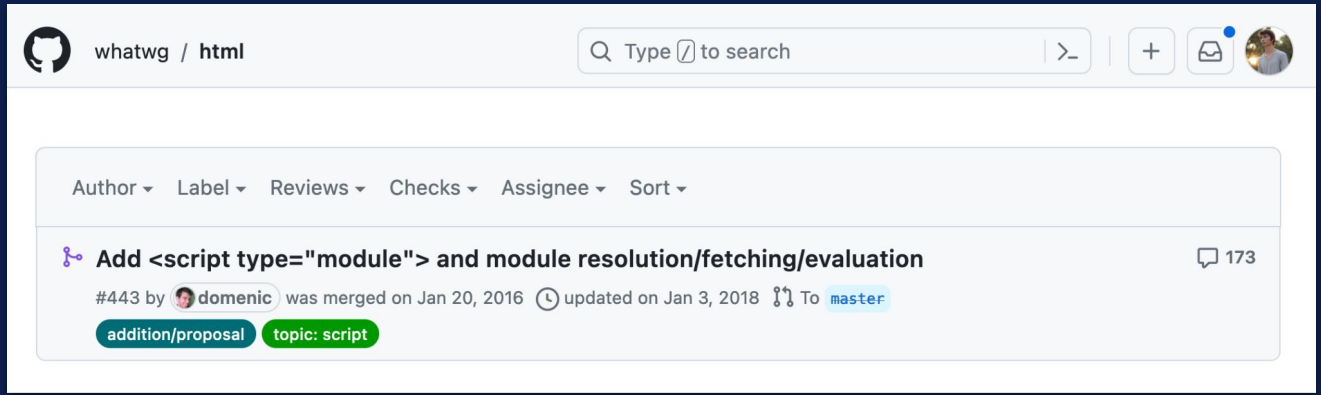

```
System.normalize = function(path) {
  if (/^text!/.test(mod)) {
    return { normalized: mod.substring(5) + ".txt", metadata: { type: 'text' } };
  }
  // fall-through for default behavior
}

System.translate = function(src, { metadata }) {
  if (metadata.type === 'text') {
    let escaped = escapeText(src);
    return `export let data = "${escaped}"`;
  }
  // fall-through for default behavior
}
```

Modules, Dave Herman, Sam Tobin-Hochstadt and Yehuda Katz, 2013
http://archives.ecma-international.org/2013/misc/2013misc4.pdf

https://nicr.dev

~~2015~~ 2016: ECMAScript Modules

ECMAScript 2015 defined *part* of the ECMAScript Modules semantics. It was just in 2016, with the HTML integration, that modules had been fully specified and implementable.



https://github.com/whatwg/html/pull/443

2019: Dynamic `import()`

Both CommonJS and AMD provided ways to dynamically require a module, but this capability was missing from the initial ECMAScript Modules version.

```
import('i18n/' + lang).then(function (i18n) {
  // modules i18n is now available for use.
});

// Since 2021, with top-level await
let i18n = await import('i18n/' + lang);
```

Today, 2023

https://nicr.dev

Today

- ECMAScript Modules are being used in production applications and as a distribution format
- CommonJS and AMD are still widely used, both by legacy and new code
- There is no clear migration path from CommonJS/AMD to ECMAScript Modules yet

What's missing from ES Modules?

A way to synchronously import dependencies only when they are actually needed.

```
// Supported by CommonJS!
exports.circ = function (radius) {
  let PI = require("./heavy-computation-pi");

  return 2 * PI * radius;
};
```

What's missing from ES Modules?

A way to easily write multiple modules in a single file.

```
// Supported by AMD!
define("pi", function () { return 3.14; });

define("math", ["pi"], function (pi) {
  return {
    circ: (radius) =>  2 * pi * radius,
  };
});
```

What's missing from ES Modules?

A way to customize the module loading process.

```
// Supported by AMD!
define("text!./doc.htm", function (docAsString) {
 // ...
});
```

What's missing from ES Modules?

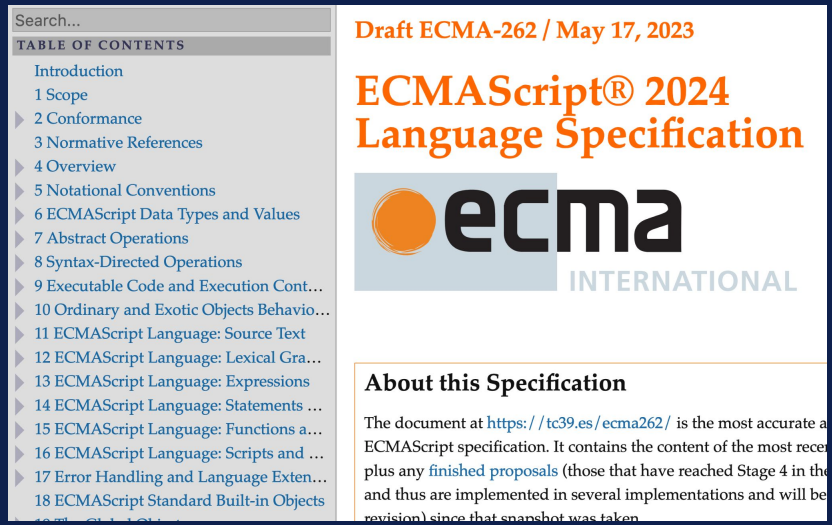
A way to properly support resources other than JavaScript.

- JSON
- WebAssembly
- ...

Modules Harmony

TC39: the JS standard committee

- TC39 designs the JavaScript language
- Made up of people from different companies, and the JS community
- Discusses and decides on new features through proposals



https://nicr.dev

Modules Harmony

- The open design space around ECMAScript Modules is huge

- ECMAScript proposals are usually self-contained and developed in isolation
  - Every proposal must be motivated on its own
  - Every proposal must be complete on its own

- Cross-proposal coordination is necessary, to ensure that the evolution of ECMAScript Modules remains consistent

Modules Harmony

January 2023 TC39 meeting

	3	60m	Problems with import assertions for module types and a possible general solution + downgrade to Stage 2 (HTML issue, slides, slides for continuation)	Nicolò Ribaudo

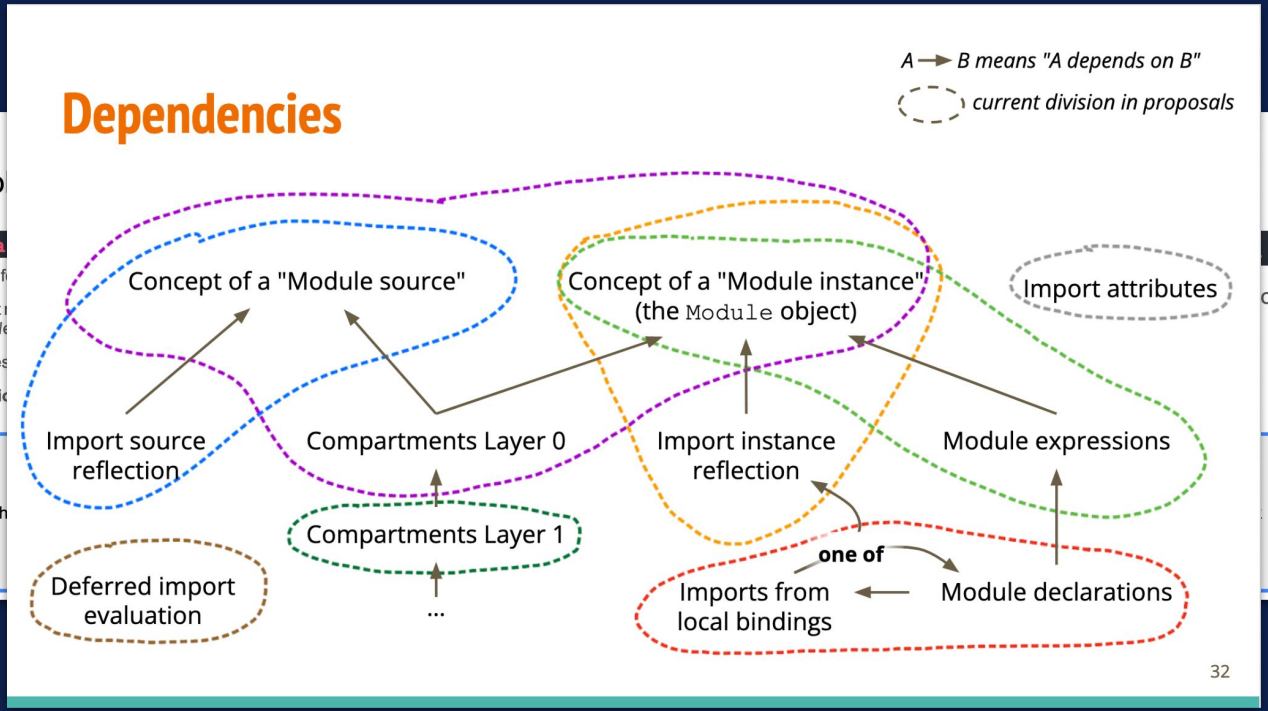
March 2023 TC39 meeting

	2	30m	Import reflection update (slides)	Luca Casonato, Guy Bedford
	2	45m	Import assertions/attributes for Stage 3 (new spec, diff, slides)	Nicolò Ribaudo

May 2023 TC39 meeting

	2	45m	Source Phase Imports for stage 3 (slides, spec)	Luca Casonato & Guy Bedford
		60m	Module Harmony: interaction semantics of the different proposals (slides)	Nicolò Ribaudo

Modules Harmony



The future

(maybe!)

Modules proposals under development

Import attributes

Source phase imports
(aka Import Reflection)

Deferred import evaluation

Custom module loading
(aka Compartments layer 0)

Module expressions

Module declarations

https://nicr.dev

Stage 3:

Import attributes

https://github.com/tc39/proposal-import-attributes

Import attributes

"Parameters for the underlying module loader"

Import attributes

"Parameters for the underlying module loader"

Please load the module
`./math.js`

Here it is! 📦

It has the these exports:

- `add`
- `sub`

And you can execute it running this:

`Evaluate() { … }`

Please load the script file
`https://x.com/math.js`

Here it is! It's an **`application/javascript`**, with these contents:

```
export let add =
  (x, y) => x + y;
export let sub =
  (x, y) => x - y;
```

Import attributes

I want this module do be loaded as a CSS file...

"Parameters for the underlying module loader"

```
import {
  styles,
} from "./m[redacted]" with {
  type: "css"
};
```

JS

Please load the module `./main.css`, the developer said that it should have

- `type: "css"`

Import attributes

"Parameters for the und

`type: "css"` means that it should be a CSS stylesheet...

JS

Please load the module **`./main.css`**, the developer said that it should have

- **`type: "css"`**

Please load the stylesheet **`https://x.com/main.css`**

Here it is! 📦

It has the these exports:

- `styles`

And you can execute it running this:

`Evaluate() { … }`

Here it is! It's a **`text/css`**, with these contents:

```
.my-class {
  color: red;
}
```

`type/css:` matches what I expected for a CSS stylesheet ✓

⚠️ Import assertions

A previous version of the proposal only allowed *asserting* that the loaded module matched a given property:

```
import styles from "./main.css" assert { type: "css" };
```

While integrating this feature in browsers, we reailzed that "assert-only" semantics didn't match what was needed so the proposal was changed:

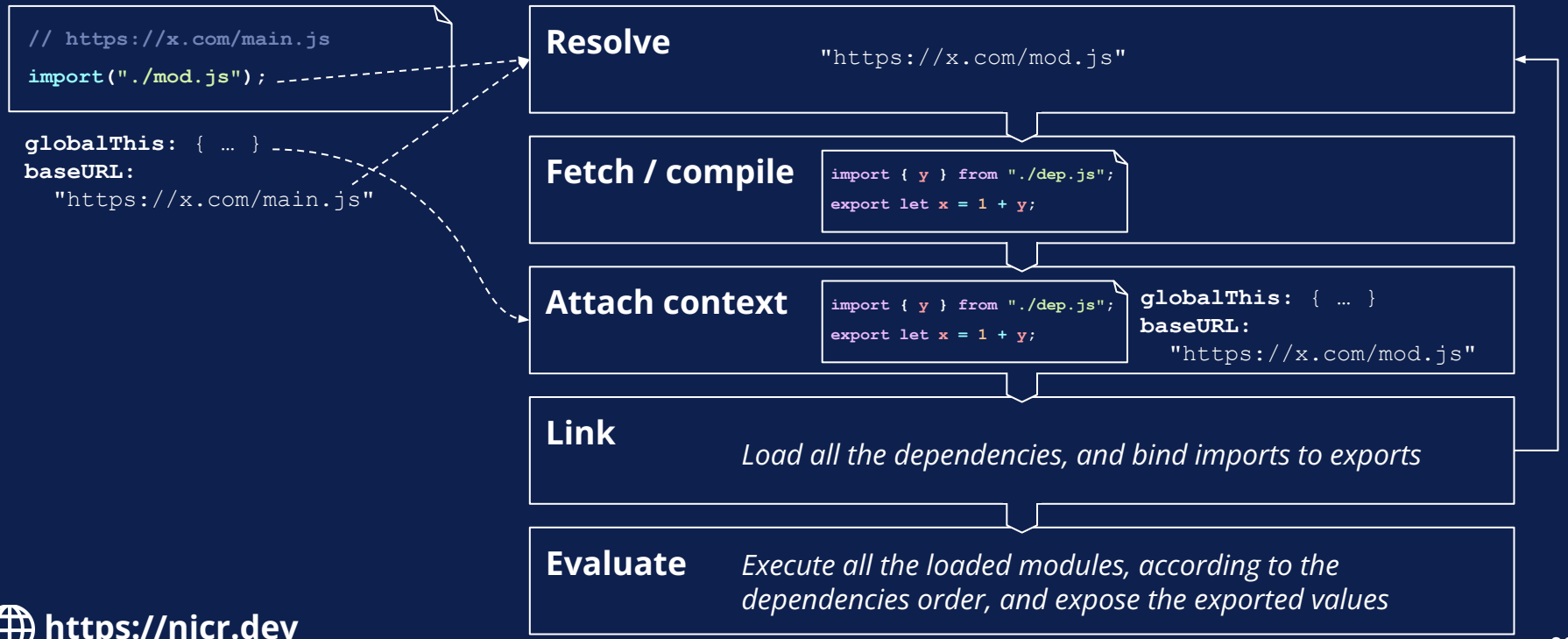
```
import styles from "./main.css" with { type: "css" };
```

Stage 2:

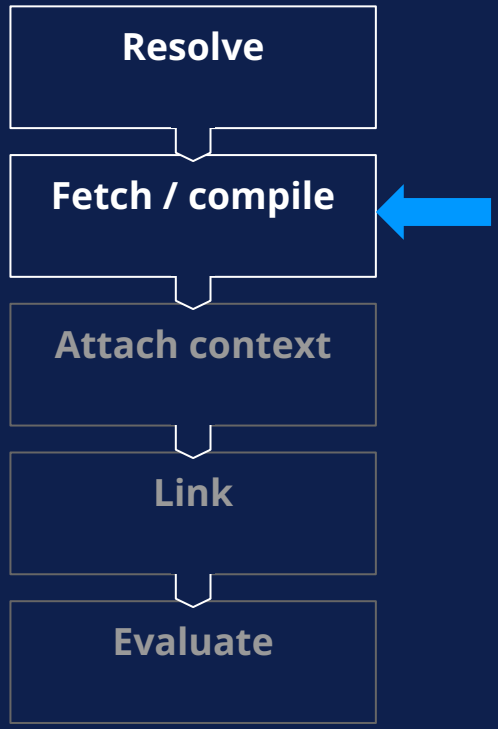
Source phase imports

https://github.com/tc39/proposal-import-reflection/

The phases of importing a module



Source phase imports



Exposing a module at an earlier phase

```
import source modSource from "./mod";
```

`modSoure` is an object representing `"./mod"`'s source, unlinked and without any execution context attached.

https://nicr.dev

WebAssembly Modules integration

Using fetch

```
const url = import.meta.resolve("./crypto.wasm");
const responseP = fetch(url);
const cryptoM =
  await WebAssembly.compileStreaming(responseP);

cryptoM instanceof WebAssembly.Module;
const cryptoI = await
WebAssembly.instantiate(cryptoM, {
  mathModule: { add: (x, y) => x + y },
});
const { md5 } = cryptoI.exports;
md5("Hello!");
// > 952d2c56d0485958336747bcdd98590d
```

Using source imports

```
import source cryptoM from "./crypto.wasm";
```

- Module is preloaded while loading all the modules
- Easily statically analyzable for bundlers
- Goes through the existing module loading content security policies (CSPs)

https://nicr.dev

WebAssembly Modules integration

Using source imports

```
import source cryptoM from "./crypto.wasm";
cryptoM instanceof WebAssembly.Module;
const cryptoI = await WebAssembly.instantiate(cryptoM, {
  mathModule: { add: (x, y) => x + y },
});
const { md5 } = cryptoI.exports;
md5("Hello!");
// > 952d2c56d0485958336747bcdd98590d
```

Fully abstracted

```
import { md5 } from "./crypto.wasm";

md5("Hello!");
// > 952d2c56d0485958336747bcdd98590d
```

- Manual linking boilerplate
- Works with any type of modules

- As simple as JS modules
- Generated Wasm modules must explicitly target the web

https://nicr.dev

Stage 1:

Deferred import evaluation

https://github.com/tc39/proposal-defer-import-eval

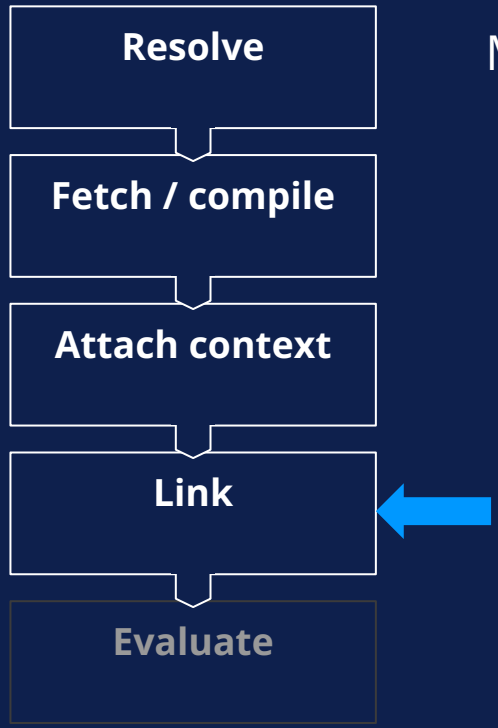
Deferred import evaluation

```
// CommonJS
exports.circ = function (radius) {
  let PI = require("./computed-pi");

  return 2 * PI * radius;
};
```

```
// ECMAScript Module
export async function circ(radius) {
  let PI = await import("./computed-pi");

  return 2 * PI * radius;
}
```

`async/await` is "viral", it forces all the callers to be asynchronous 😕

Deferred import evaluation



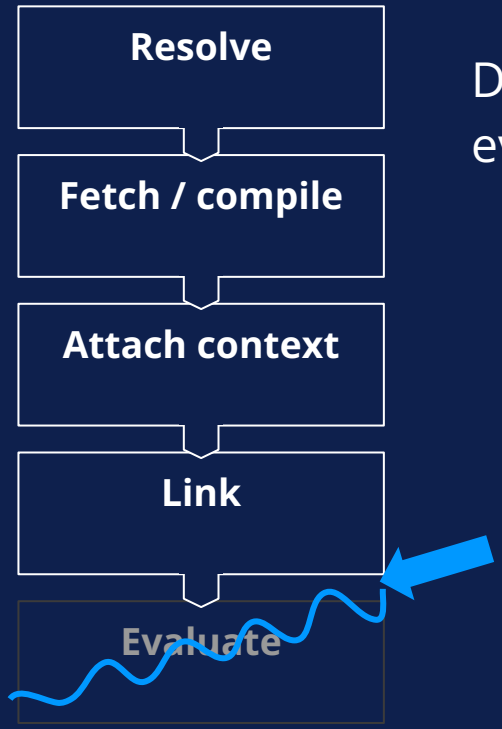
Module *loading* must be asynchronous. Can we still avoid some initialization cost?

```
import defer * as mod from "./computed-pi.js"
export function circ(radius) {
  return 2 * mod.PI * radius;
}
```

`mod` is an import namespace object that triggers the evaluation of the corresponding module only when accessing its exports.

https://nicr.dev

Blurring the line: top-level `await`



Due to top-level `await`, some modules cannot be evaluated synchronously.

```js
// a.js
import defer * as b from "./b.js";
console.log("Eval A");
```

```js
// b.js
import "./c.js";
console.log("Eval B");
```

```js
// c.js
await aPromise;
console.log("Eval C");
```

- Since `c.js` is *asynchronous*, its evaluation cannot be optimized await and it's evaluated **eagerly**.
- Later, when accessing `b.something`, only `b.js` still needs to be evaluated.

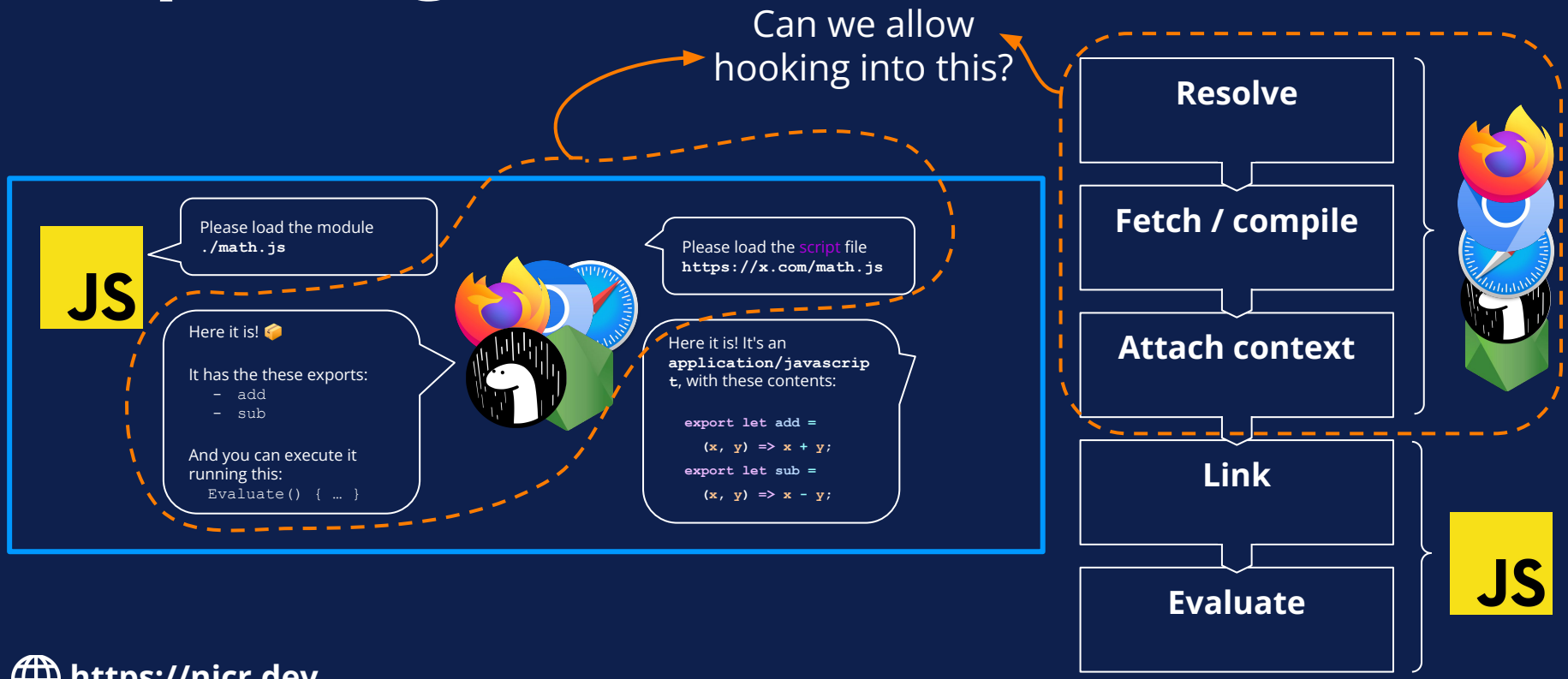
https://nicr.dev

Stage 1:

Custom module loading

https://github.com/tc39/proposal-compartments

Importing a module



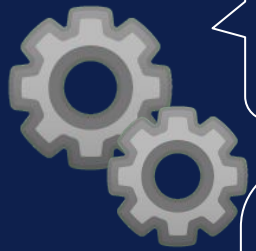
https://nicr.dev

Browsers: Service Workers

Fetch / compile

Please load the script file
`https://x.com/math.ts`

The service worker receives the resolved URL…



Please load the script file
`https://x.com/math.ts`

Here it is! It's an
`application/javascript`,
with these contents:

```
export let add =
  (x, y) => x + y;
export let sub =
  (x, y) => x - y;
```

Here it is! It's an
`application/typescript`,
with these contents:

```
export let add =
  (x: num, y: num) => x + y;
export let sub =
  (x: num, y: num) => x - y;
```

… and returns the corresponding source.

https://nicr.dev

Browsers: Service Workers

```
// service-worker.js
addEventListener("fetch", function (event) {
  if (event.request.url.endsWith(".ts")) {
    event.respondWith(
      fetch(event.request).then(async response => {
        let { code } = await babel.transform(await response.text(), babelConfig).code;
        return new Response(code, {
          headers: { ...response.headers, "content-type": "text/javascript" }
        });
      })
    );
  } else {
    event.respondWith(fetch(event.request));
  }
});
```


https://developer.mozilla.org/en-US/docs/Web/API/ServiceWorkerGlobalScope/fetch_event

Node.js: `--experimental-loader`

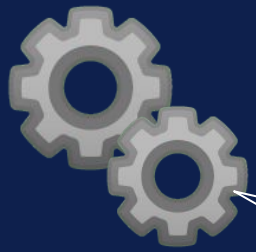
Please resolve `./math.ts` from `file:///proj/main.js`

It resolves to `file:///proj/math.ts`

Please load the file `file:///proj/math.ts`

Here it is! It's a `module` file, with these contents:

```
// ...
```



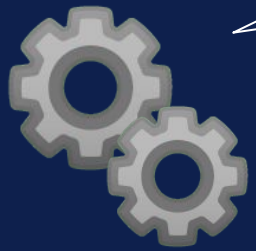
Resolve

...

...

...

...



Fetch / compile

https://nicr.dev

https://nodejs.org/api/esm.html#loaders

Proposal: custom module loading

Please load the module
`./math.js`

Here it is! 📦

It has the these exports:

- `add`
- `sub`

And you can execute it running this:

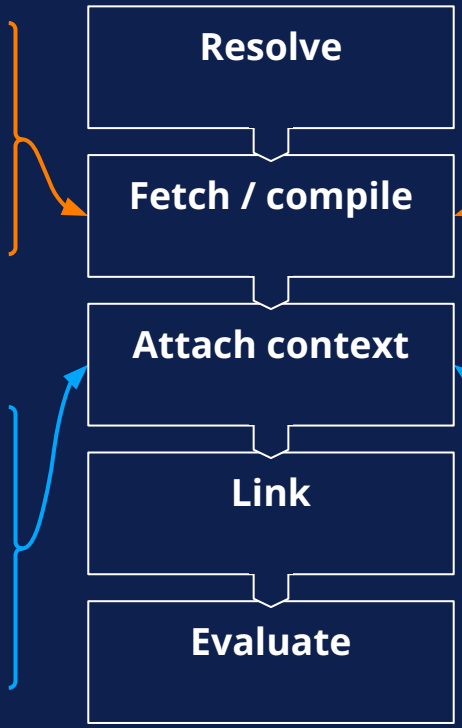
`Evaluate() { … }`

```
new Module(source, {
  async importHook(specifier) {
    const url = resolveURL(specifier);
    const source = await fetchSource(url);
    return new Module(source, hooks);
  },
});
```

Resolve → **Fetch / compile** → **Attach context**

Module and ModuleSource

```
import source wasmS from "./mod.wasm";
wasmS instanceof WebAssembly.Module;

import source jsS from "./mod.js";
jsS instanceof ???;
```




```
let source = WebAssembly.compile(bytes);
let source = new ModuleSource(`
  import { x } from "dep";
  console.log(x);
`);
```

```
import module wasmM from "./mod.wasm";
wasmM instanceof Module;

import module jsM from "./mod.js";
jsM instanceof Module;
await import(jsM);
```

```
let module = new Module(source, {
  async importHook() { /* ... */ },
  importMeta: { /* ... */ },
  // Any other context
  baseURL: "https://...",
});
await import(module);
```

https://nicr.dev

Stage 2:

Module expressions

Module expressions

Modules would now hava a first-class representation in the language (like functions):

```
fun instanceof Function;
```

```
mod instanceof Module;
```

You can create them with the respective constructors:

```
let fun = new Function(
  "x",
  "return x + 1;"
);
```

```
let mod = new Module(
  new ModuleSource(`
    export default 7;
  `)
);
```

... or with static syntax:

```
let fun = function (x) {
  return x + 1;
};
```

```
let mod = module {
  export default 7;
};
```

Module expressions!

https://nicr.dev

Module expressions

You can create them with the respective constructors:

```
let fun = new Function(
  "x",
  "return x + 1;"
);

let mod = new Module(
  new ModuleSource(`
    export default 7;
  `)
```

... or with static syntax:

```
let fun = function (x) {
  return x + 1;
};

let mod = module {
  export default 7;
};
```

Module expressions!

And you can later execute them:

```
fun();

await import(mod);
```

https://nicr.dev

Module expressions

You can create them with the respective constructors:

```
let fun = new Function(
  "x",
  "return x + 1;"
);

let mod = new Module(
  new ModuleSource(`
    export default 7;
  `)
```

... or with static syntax:

```
let fun = function (x) {
  return x + 1;
};

let mod = module {
  export default 7;
};
```

Module expressions!

Functions have a *declaration* form:

```
function fun(x) {
  return x + 1;
}
```

???

https://nicr.dev

Module expressions use cases

Co-locating code to be executed in different threads

```
let worker = new Worker("/modules-runner");
worker.postMessage(module {
  export default function () {
    // Do some expensive work!
  }
});
worker.addEventListener("message", ({ data }) => {
  console.log("Result", data + 2);
});
```

Logically linked code can live in the same file

Improved ergonomics for custom loaders

```
import source modSource from "./my-file.js";
// Mock access to the file system
async function importHook(specifier) {
  if (specifier === "fs") {
    return module {
      export const readFile = () => "Hello!";
    };
  }
  return import(resolve(specifier));
}

import(new Module(modSource, { importHook }));
```

https://nicr.dev

Stage 2:

Module declarations

Module declarations

You can create them with the respective constructors:

```
let fun = new Function(
  "x",
  "return x + 1;"
);

let mod = new Module(
  new ModuleSource(`
    export default 7;
  `)
```

... or with static syntax:

```
let fun = function (x) {
  return x + 1;
};

let mod = module {
  export default 7;
};
```

Functions and modules have a *declaration* form:

```
function fun(x) {
  return x + 1;
}

module Mod {
  export default 7;
}
```

Module declarations!

https://nicr.dev

Bundling primitives in ECMAScript

What's missing from ES Modules?

A way to easily write multiple modules in a single file.

```
// Supported by AMD!
define("pi", function () { return 3.14; });

define("math", ["pi"], function (pi) {
  return {
    circ: (radius) =>  2 * pi * radius,
  };
});
```

@NicoloRibaudo

24

https://nicr.dev

Module declarations

```
// bundle.amd.js

define("pi", function () {
  return 3.14;
});
define("math", ["pi"], function (pi) {
  return {
    circ(radius) {
      return 2 * pi * radius;
    }
  };
});
```

```
// bundle.esm.js

module PI {
  export default 3.14;
}
module Math {
  import pi from PI;
  export function circ(radius) {
    return 2 * pi * radius;
  }
}
```

Module declarations

They can be imported, exported, nested, and passed around.

```
// vendor.bundle.js

export module lodash {
  module Internal { /* ... */ }
  module Get {
    import { something } from Internal;
    export function get(obj, path) { /* ... */ }
  }
  export { get } from Get;
  // ...
}
```

```
// vendor.external.js

export module Preact from "https://example.com/preact@10.13.2";
```

```
// main.js

import { lodash } from "./vendor.bundle.js";
import { Preact } from "./vendor.external.js";

import { get, set } from lodash;
import { h } from Preact;

get({ a: 1 }, "a");
```

Modules proposals under development

Import attributes

Source phase imports
(aka Import Reflection)

Deferred import evaluation

Custom module loading
(aka Compartments layer 0)

Module expressions

Module declarations

https://nicr.dev

When is all of this coming?

Some proposals are approaching their final shape,
but many are still in their exploration phase.