Node & ECMAScript Modules

Jason Pollman
jason.pollman@passportinc.com

https://github.com/jasonpollman/esm-examples

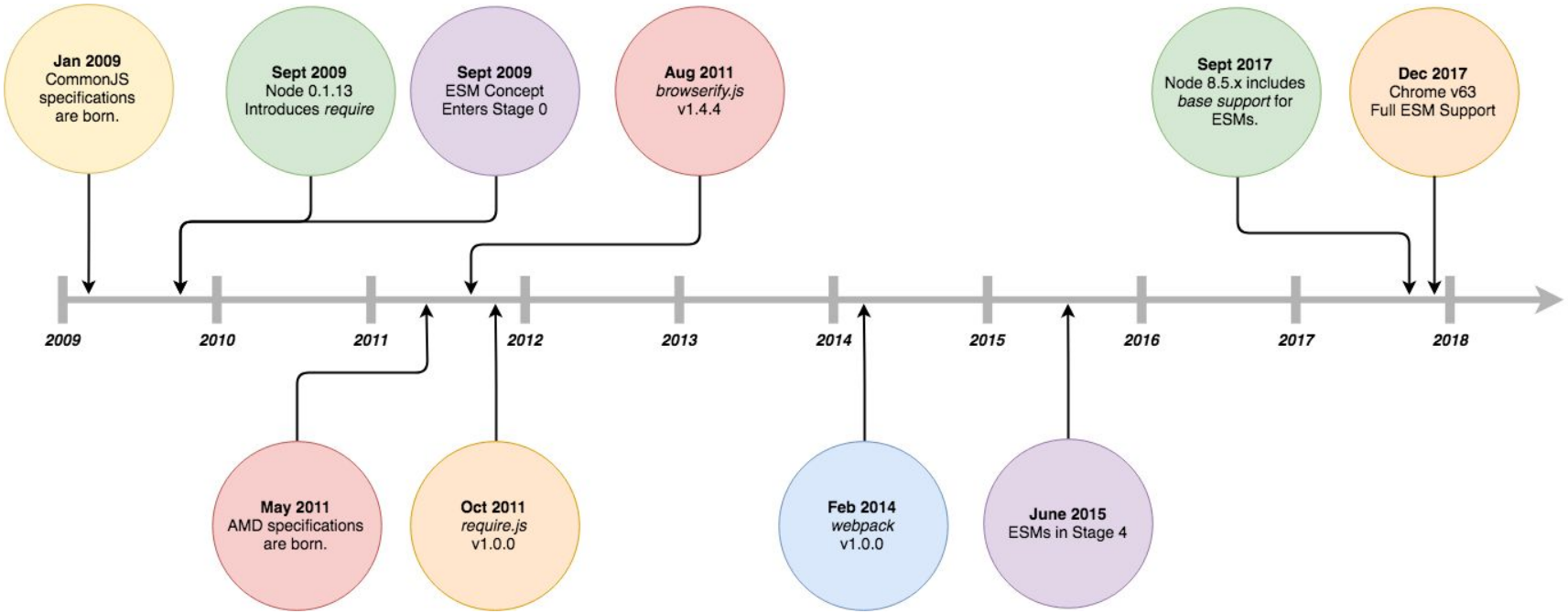
What are ECMAScript Modules (ESMs)?

The *standard* JavaScript module system.
The ES2015 spec defined native JavaScript modules—including specs for importing and exporting symbols in a standardized way. This means a single module system that works in both browsers and in node.js.

Prior to ES2015, JavaScript had no built-in module system.
Workarounds like *CommonJS* (CJS) and *Asynchronous Module Definition* (AMD) were developed in lieu of a native module system.

Node's *require* is a CommonJS variant.
Node uses a CJS like module system that is *synchronous*.

Brief History of JavaScript Module Systems



Why not just use CJS or AMD?

CJS

- **Easy Syntax**
  Requires no “special module wrappers”.
- **Synchronous**
  Problematic for browsers: exports are unknown until the module completely executes, so you can’t load modules asynchronously.
- **Dynamic & Conditional Loading**
  Inhibits asynchronous module loading.

AMD

- **Hard to use Syntax**
  Code is wrapped in a “define” wrapper.
- **Asynchronous**
  Can fetch multiple imports at a time since all imports are known before code is executed.
- **Static Loading**
  No code is executed until all module dependencies have been resolved.
- **Not Compatible with Node.js**

ESM Export Syntax

```
// Export the default export
export default function () { ... }

// Default export can be anything,
// but you can only have *one* default.
export default 5;

// Export *named* exports
export function bar() { ... }
const x = { value: 'value' };
const y = 'hello world!';
export { x, y };
```

ESM Import Syntax

```
// Import only the default export
import foo from './foo.mjs';

// Import *all* exports (including default)
import * as foo from './foo.mjs';

// Import *named* exports
import { thing } from './foo.mjs';
import { thing as thingy } from './foo.mjs';

// Import default and named
import foo, { thing } from './foo.mjs';
```

I've been using ESMs for a long time!

Probably not.

(babel/ts transpiles your code to CJS).

```
export default function foo() {}
export function bar() {}

import foo from './foo';
import { bar } from './bar';
```



```
exports.default = function foo() {}
exports.bar = function () {}

const foo = require('./foo').default;
const bar = require('./foo').bar;
```

node --experimental-modules ./my-module.mjs

ESMs are *still* experimental, you must “opt in” to using them.

.mjs vs .js

For backward compatibility, ESMs use the *.mjs* file extension.

In node, files with the *.js* extension will be treated as CJS modules.

npm run basic

Basic usage of using ESM imports/exports.

With all this .mjs stuff why bother?
require works just fine for me!

[interoperability]

One, module system to rule them all!

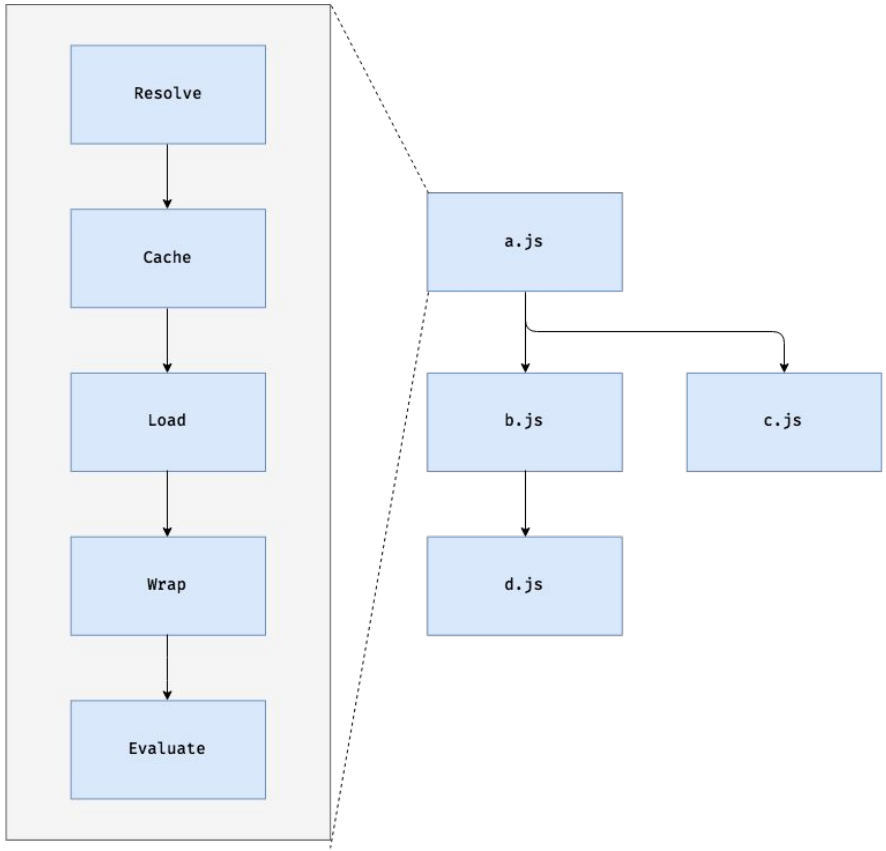
static vs. dynamic

ESM is a **static** module system, CJS is **dynamic**.

```
require('./foo')
```

What happens when we *require*?

CJS Module Loading




```
// a.js
const b = require('./b');
const c = require('./c');

// b.js
const d = require('d');
```

1. Resolve/Cache/Load/Wrap/Evaluate A
2. Resolve/Cache/Load/Wrap/Evaluate B
3. **Resolve/Cache/Load/Wrap/Evaluate D**
4. Resolve/Cache/Load/Wrap/Evaluate C

CJS Module Loading

```
Module._extensions['.js'] = function(module, filename) {
  var content = fs.readFileSync(filename, 'utf8');
  module._compile(content, filename);
};
```

CJS Module Loading

```
var wrapper = Module.wrap(content);

var compiled = vm.runInThisContext(wrapper, ...);

result = compiled.call(this.exports,
  this.exports, require, this, filename, dirname
);
```

CJS Module Loading

```
Module.wrapper = [
  '(function (exports, require, module, __filename, __dirname) { ',
  '\n});'
];

Module.wrap = function(script) {
  return Module.wrapper[0] + script + Module.wrapper[1];
};
```

```
import { foo } from './foo';
```

What happens when we *import*?

ESM Module Loading

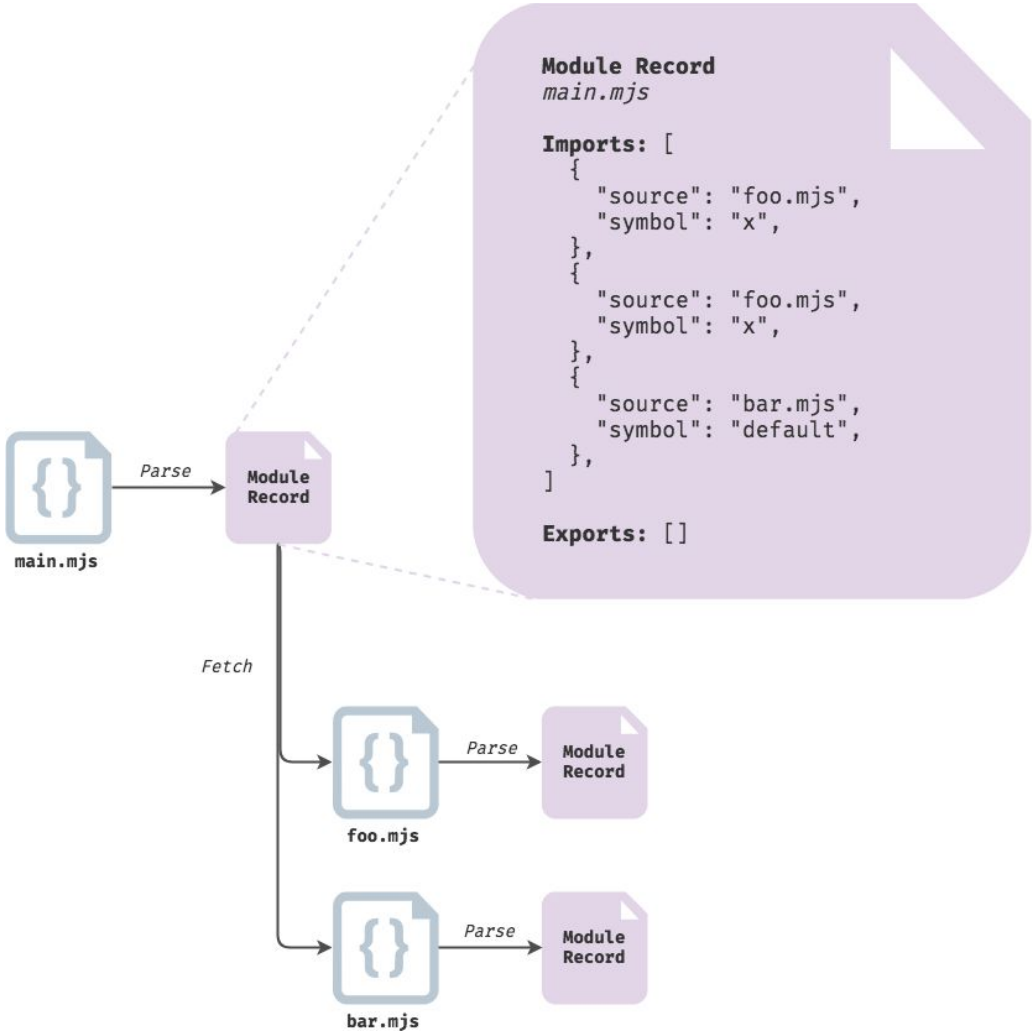
ESM Module Loading

Construction

```
// main.mjs
import { x, y } from './foo.mjs';
import bar from './bar.mjs';

// foo.mjs
const x = 'x';
const y = 'y';
export { x, y };

// bar.mjs
export default () => { ... };
```



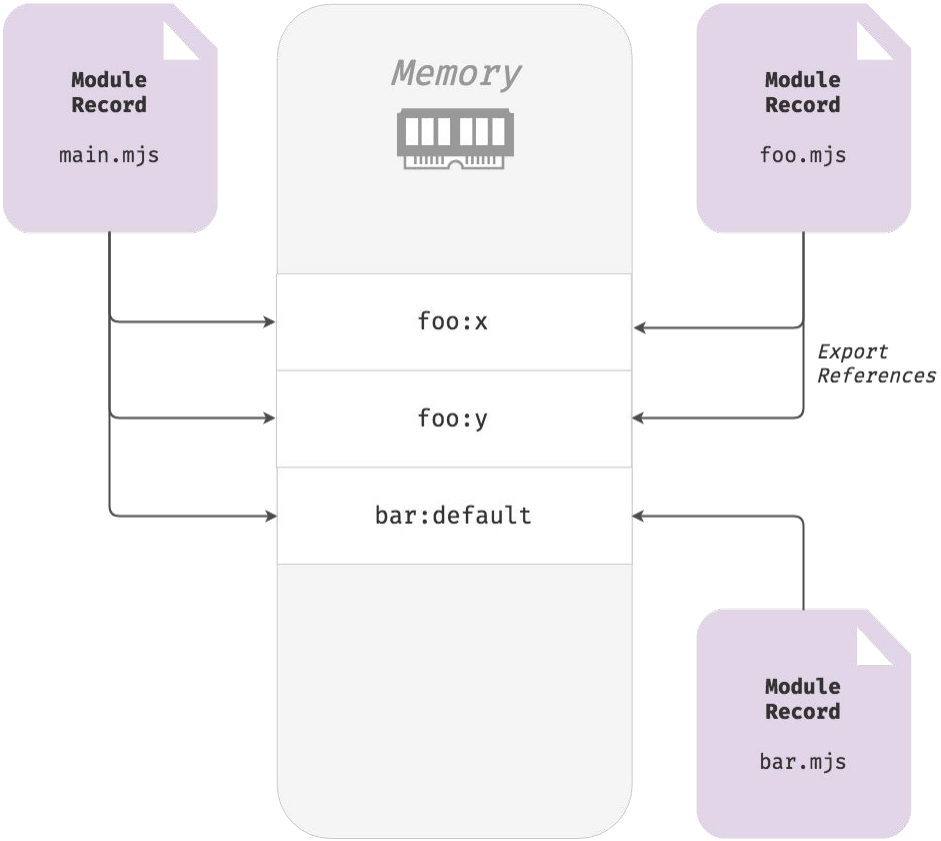
ESM Module Loading

Instantiation

```
// main.mjs
import { x, y } from './foo.mjs';
import bar from './bar.mjs';

// foo.mjs
const x = 'x';
const y = 'y';
export { x, y };

// bar.mjs
export default () => { ... };
```



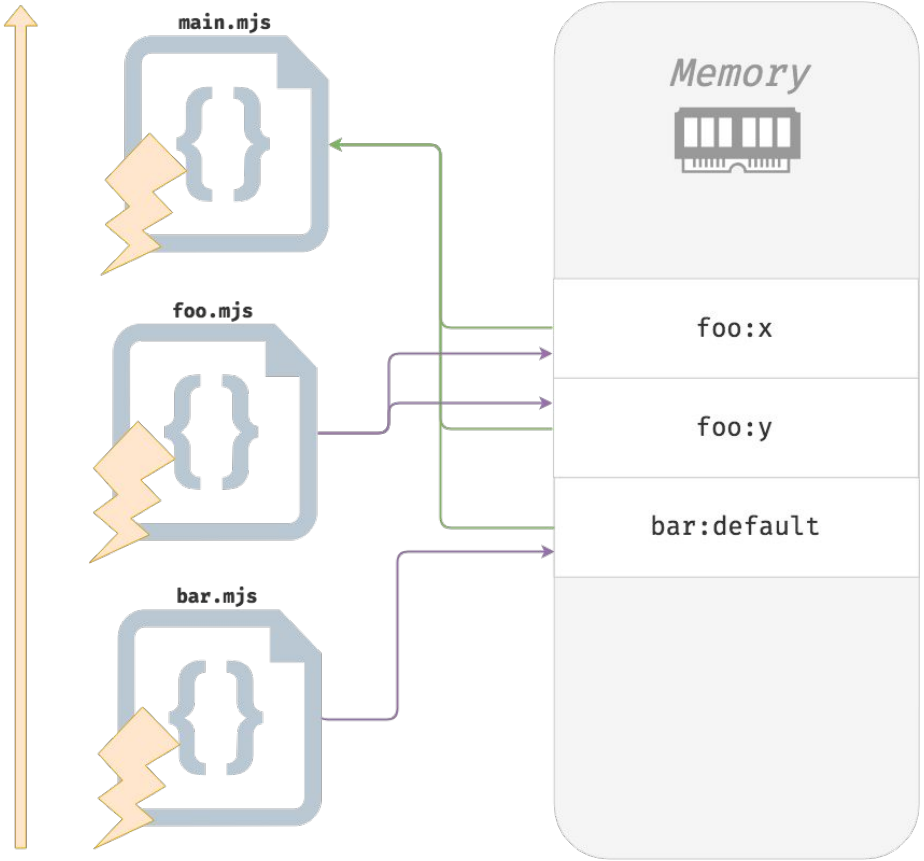
ESM Module Loading

Evaluation

```
// main.mjs
import { x, y } from './foo.mjs';
import bar from './bar.mjs';

// foo.mjs
const x = 'x';
const y = 'y';
export { x, y };

// bar.mjs
export default () => { ... };
```



ESM Module Loading

```
translators.set('esm', async (url) => {
  const source = await readFileAsync(new URL(url));

  return {
    // ModuleWrap comes from node/src/module_wrap.cc
    // It's a wrapper around V8's internal Module.
    module: new ModuleWrap(source, url),
    ...
  };
});
```

ESM Module Loading

```
// Loader#import
async import(specifier, parent) {
  // Gets or creates a ModuleJob instance.
  // ModuleJob#constructor creates promises
  // for linking all dependent modules.
  const job = await this.getModuleJob(specifier, parent);

  const module = await job.run();
  return module.namespace();
}
```

ESM Module Loading

```
// ModuleJob#run
async run() {
  // ModuleJob#instantiate waits for all
  // dependent modules to be linked and returns
  // the native ESM module from V8.
  const module = await this.instantiate();

  // module.evaluate(timeout, breakOnSigint);
  module.evaluate(-1, false);
  return module;
}
```

npm run missing

Demonstrates ESM vs CJS handling of missing dependencies

npm run circular

Shows that circular dependencies are possible with ESMs.

npm run hoisted

Shows that *import* has “hoisting like” behavior.

What about dynamic imports?

Calling ***import(...)*** will return a promise.

```
import('./my-module.mjs').then(exports => { ... });
```

npm run dynamic

Basic usage of *import()* syntax

Custom Loaders

Custom loaders allow you to define how modules are loaded.

node --experimental-modules --loader ./my-loader.mjs ./my-module.mjs

...smells like require.extensions...

npm run custom

Shows how to use custom module loaders in Node.js

Can I use?

- Support in all major browsers (except IE).
- Stage 4 means the API won't change.
- Dynamic *import()* is in stage 3.

- Wait for non-experimental support in Node.
- *node_modules* don't resolve in the browser; wait for module resolution proposal.
- Until http2 is widely implemented, the performance of bundling systems will outperform modules.

```
process.emit('questions');
```