

How to Write Module-Based JavaScript with RequireJS and the AMD (Asynchronous Module Definition) Pattern

Eric O'Hanlon

Web Developer
Columbia University

elo2112@columbia.edu
@elohanlon on Github
and Samvera Slack

Requirement: Google Chrome 62

- Please download Google Chrome if you don't already have it
- If you do have Google Chrome, make sure that you're on version 62
 - Righthand three-vertical-dot menu -> Help -> About Google Chrome

Important Links

<https://wiki.duraspace.org/display/samvera/Code+of+Conduct>

<https://wiki.duraspace.org/display/samvera/Anti-Harassment+Policy>

What is Asynchronous Module Definition (AMD)?

"The Asynchronous Module Definition (**AMD**) API specifies a mechanism for **defining modules** such that the module **and its dependencies** can be **asynchronously loaded**. This is particularly well suited for the browser environment where synchronous loading of modules incurs performance, usability, debugging, and cross-domain access problems."

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



What is Asynchronous Module Definition (AMD)?

"It is unrelated to the technology company AMD and the processors it makes."

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

What is Asynchronous Module Definition (AMD)?


- **Asynchronous:** Non-blocking loading
- **Module:** Decoupled piece of functionality
- **Definition:** Established in a registered and retrievable place

Or put another way...

Ability to encapsulate a piece of code into a useful unit, register its capability, and export a value.

And be able to refer to those units of code from other parts of our code.

- <http://requirejs.org/docs/whyamd.html>



Common Problems With Non-Module-Based JavaScript

- Difficulty managing script tag load order
- Polluting the global namespace
- Variable Name Conflicts

Difficulty Managing Script Tag Load Order

```
<body>
  <script src="../../test/vendor/json2.js"></script>
  <script src="../../test/vendor/jquery.js"></script>
  <script src="../../test/vendor/underscore.js"></script>
  <script src="../../backbone.js"></script> <!-- must be after underscore -->
  <script src="../../backbone.localStorage.js"></script>
  <script src="todos.js"></script>
</body>
```

"As JavaScript development gets more and more complex, dependency management can get cumbersome. Refactoring is also impaired: where should newer dependencies be put to maintain proper order of the load chain?"

- <https://autho.com/blog/javascript-module-systems-showdown/>

"The dependencies are very weakly stated: the developer needs to know the right dependency order. For instance, The file containing Backbone cannot come before the jQuery tag."

- <http://requirejs.org/docs/whyamd.html>

Polluting the Global Namespace

"As variables lose scope, they will be eligible for garbage collection. If they are scoped globally, then they will not be eligible for collection until the global namespace loses scope."

```
var arr = [];  
for (var i = 0; i < 2003000; i++) {  
  arr.push(i * i + i);  
}
```

"Adding this to your global namespace...should add 10,000 kb of memory usage...which will not be collected."

- <https://stackoverflow.com/questions/8862665/what-does-it-mean-global-namespace-would-be-polluted>

Polluting the Global Namespace

"Whereas having that same code in a scope which goes out of scope like this... will allow [the array] to lose scope after the closure executes and be eligible for garbage collection."

```
(function(){  
  var arr = [];  
  for (var i = 0; i < 2003000; i++) {  
    arr.push(i * i + i);  
  }  
})();
```

- <https://stackoverflow.com/questions/8862665/what-does-it-mean-global-namespace-would-be-polluted>

Variable Name Conflicts

"Without some sort of *encapsulation* it is a matter of time before two *modules* conflict with each other. This is one of the reasons elements in C libraries usually carry a prefix."

- <https://autho.com/blog/javascript-module-systems-showdown/>

e.g. jQuery and Prototype libraries both use "\$" variable



AMD Solves All of These Problems!

And it provides a great framework for breaking up your code into manageable and reusable modules.

Brief JavaScript Module History

- In 2009, the CommonJS project formed with the intention of developing a specification for JavaScript usage outside of the web browser.
- CommonJS defined a useful module pattern with automatic dependency loading and circular reference handling, but loading of dependencies was synchronous.
- JavaScript browser community branched off from CommonJS and formed AMD in order to meet asynchronous environment needs.
- The server-side Node.js community embraced CommonJS.

RequireJS

- RequireJS is the most popular among several different AMD module loaders.
- Other loaders include:
 - Isjs (<https://github.com/zazl/Isjs>)
 - Dojo (<http://dojotoolkit.org>)
 - curl (<https://github.com/cujojs/curl>)
 - not related to the cURL data transfer utility)



How RequireJS Works

Two major methods to understand: **define** and **require**

How RequireJS Works

```
define(  
  module_id, // optional  
  [dependencies], // optional  
  definition function(*dependencies) // function for instantiating the module or object  
);
```

How RequireJS Works

```
define('myModule',
  ['foo', 'bar'],
  // module definition function
  // dependencies (foo and bar) are mapped to function parameters
  function ( foo, bar ) {
    // return a value that defines the module export
    // (i.e the functionality we want to expose for consumption)

    // create your module here
    var myModule = {
      doStuff : function(){
        console.log('Yay! Stuff');
      }
    }

    return myModule;
  });
```

How RequireJS Works

```
require(  
  [dependencies], // optional  
  function(*dependencies) // whatever code you want to execute  
);
```

How RequireJS Works

```
// 'foo' and 'bar' are two external modules
// In this example, the 'exports' from the two modules loaded are passed as
// function arguments to the callback (foo and bar)
// so that they can similarly be accessed
//If foo.js or bar.js call define(), then this function is not fired until
// both dependencies have loaded, and the foo and bar arguments will hold
//the module value for "helper/util".
```

```
require(['foo', 'bar'], function ( foo, bar ) {
    // rest of your code here
    foo.doSomething();
    bar.doSomethingElse();
});
```

How RequireJS Works

Application Entry Point

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Sample Project</title>
    <!--When require.js loads it will inject another script tag (with async attribute) for scripts/main.js-->
    <!--Since the load is asynchronous, you can't assume that the DOM will have fully loaded yet! -->
    <script data-main="scripts/main" src="scripts/require.js"></script>
  </head>
  <body>
    <h1>My Sample Project</h1>
  </body>
</html>
```

How RequireJS Works

```
//main.js  
requirejs(["helper/util"], function(util) {  
    ...  
    ...  
});
```

Basic AMD Setup

```
//helper/util.js  
define(function () {  
    function Util() {  
        ...  
        ...  
    }  
  
    return Util;  
});
```

Deploying RequireJS App To Production

- Works as is. No build / pre-processing required.
- Possible to optimize with RequireJS Optimizer
 - "Once you are finished doing development and want to deploy your code for your end users, you can use the [optimizer](http://requirejs.org/docs/optimization.html) to combine the JavaScript files together and minify it. In the example above, it can combine main.js and helper/util.js into one file and minify the result."
 - <http://requirejs.org/docs/optimization.html>

RequireJS API

- We're covering the basics today, but there's even more to RequireJS!
- The full RequireJS API can be found here: <http://requirejs.org/docs/api.html>



Best Way To Learn

Do it yourself!

Let's Make An App

Race Simulator



I love JavaScript!



We'll Start Here

<https://github.com/elohanlon/race-simulator/tree/part-01>

Follow The Diffs For Each Step

<https://github.com/elohanlon/race-simulator/compare/part-01...part-02>

<https://github.com/elohanlon/race-simulator/compare/part-02...part-03>

<https://github.com/elohanlon/race-simulator/compare/part-03...part-04>

...

<https://github.com/elohanlon/race-simulator/compare/part-11...part-12>

Confession

- Interest in RequireJS is winding down, and it will be replaced by native browser support for ECMAScript 6 modules in the near future.
- But what we've learned won't go to waste! The pattern is very similar.
- In part 12 of our workshop, we'll be converting our RequireJS app to use ES6 modules instead. Easy change.

What is ES6 (ECMAScript 6)?

ECMAScript is a standard which is specified by Ecma International.

JavaScript is the most well-known implementation of this standard.

Other lesser known implementations include...

- JScript (Microsoft's variation on ECMAScript, implemented in Internet Explorer)

- ActionScript (the language used for programming in Flash)

What is ES6 (ECMAScript 6)?

"The **6th** edition, officially known as ECMAScript 2015, was finalized in June 2015. This update adds significant new syntax for writing complex applications, including classes and modules, but defines them semantically in the same terms as ECMAScript 5 strict mode..."

...and it adds some other cool things that we won't talk about today, but check it out!

Important to know: "Browser support for ES2015 is still incomplete. However, ES2015 code can be transpiled into ES5 code, which has more consistent support across browsers.

Transpiling adds an extra step to build processes whereas **polyfills** allow adding extra functionalities by including another JavaScript file."

- https://en.wikipedia.org/wiki/ECMAScript#6th_Edition_-_ECMAScript_2015

Native ES6 Module Support in Modern Browsers

Not enabled by default in Firefox 56 (current version)

Requires enabling of "dom.moduleScripts.enabled" setting in about:config

Enabled by default **for the first time** in Chrome 62 (current version)

Chrome 62 released a few weeks ago. Previous version required enabling "Experimental Web Platform" in chrome:flags.

Enabled by default **for the first time** in Edge 16 (current version)

Edge 16 released a little over a month ago.

What was that about "transpiling"?

"Browser support for ES2015 is still incomplete. However, ES2015 code can be **transpiled** into ES5 code, which has more consistent support across browsers. **Transpiling** adds an extra step to build processes whereas **polyfills** allow adding extra functionalities by including another JavaScript file."

- https://en.wikipedia.org/wiki/ECMAScript#6th_Edition_-_ECMAScript_2015

Transpiling ES6 code into ES5 code means:

Converting ES6 keywords and code into alternate syntax that works the same way, but uses only ES5-compatible syntax.

Converted syntax is generally more complex and verbose, but it works.

What was that about "transpiling"?

See Babel transpiler as an example: <https://babeljs.io>
(or <http://babeljs.io/repl> and check off "es2015" on left)

Let's try converting this ES6 class declaration to ES5-compatible syntax:

```
class MyClass {  
  constructor(str) {  
    this.str = str;  
  }  
  getStr() {  
    return this.str;  
  }  
  static staticMethod() {  
    console.log("so static!");  
  }  
}
```

ES6 Modules

	Scripts	Modules
HTML element	<code><script></code>	<code><script type="module"></code>
Default mode	non-strict	strict
Top-level variables are	global	local to module
Value of <code>this</code> at top level	<code>window</code>	<code>undefined</code>
Executed	synchronously	asynchronously
Declarative imports (<code>import</code> statement)	no	yes
Programmatic imports (Promise-based API)	yes	yes
File extension	<code>.js</code>	<code>.js</code>

16.6.1.1 Scripts

ES6 Modules

Each module is a piece of code that is executed once it is loaded.

In that code, there may be declarations (variable declarations, function declarations, etc.).

By default, these declarations stay local to the module.

You can mark some of them as exports, then other modules can import them.

A module can import things from other modules. It refers to those modules via *module specifiers*, strings that are either:

Relative paths ('../model/user'): these paths are interpreted relatively to the location of the importing module. The file extension .js can usually be omitted.

Absolute paths ('/lib/js/helpers'): point directly to the file of the module to be imported.

Names ('util'): What modules names refer to has to be configured.

Modules are singletons. Even if a module is imported multiple times, only a single “instance” of it exists.

This approach to modules avoids global variables, the only things that are global are module specifiers.

ES6 Modules

"Modules are deferred, and only run after a document is loaded"

"The import and export statements can only appear at the top-level of a file and cannot include variables, so the full import graph can be determined trivially—a bit like C/C++ `#include` or Python `import`"

"All module code is run in [strict mode](#)—aka 'use strict';"

From: <https://medium.com/dev-channel/es6-modules-in-chrome-canary-m6o-ba588dfb8ab7>

Good for AMD People and CommonJS People

- AMD and CommonJS split because one group wanted asynchronous and the other wanted synchronous.
- ES6 modules are loaded asynchronously in the browser, and synchronously in Node.js.
- Same syntax, everybody wins!

ES6 Import/Export Syntax

There can be multiple *named exports*:

```
//----- lib.js -----  
export const sqrt = Math.sqrt;  
export function square(x) {  
    return x * x;  
}  
  
export function diag(x, y) {  
    return sqrt(square(x) + square(y));  
}  
  
//----- main.js -----  
import { square, diag } from 'lib';  
console.log(square(11)); // 121  
console.log(diag(4, 3)); // 5
```

And you can also import the complete module:

```
//----- main.js -----  
import * as lib from 'lib';  
console.log(lib.square(11)); // 121  
console.log(lib.diag(4, 3)); // 5
```

Or there can be a single *default export*. For example, a function:

```
//----- myFunc.js -----  
export default function () { ... } // no  
semicolon!  
  
//----- main1.js -----  
import myFunc from 'myFunc';  
myFunc();
```

Or a class:

```
//----- MyClass.js -----  
export default class { ... } // no semicolon!  
  
//----- main2.js -----  
import MyClass from 'MyClass';  
const inst = new MyClass();
```


Let's update our app!



I love ES6!

And that's it!

Questions?
Comments?



Eric O'Hanlon
elo2112@columbia.edu
@elohanlon on Github
and Samvera Slack