[**CommonJS: Why and How**](https://0fps.net/2013/01/22/commonjs-why-and-how/)

Posted on [January 22, 2013](https://0fps.net/2013/01/22/commonjs-why-and-how/)by [mikolalysenko](https://0fps.net/author/mikolalysenko/)

As I said [last time](https://0fps.wordpress.com/2013/01/15/new-years-resolution-post-more-stuff-on-npm/), I want to start moving more of my stuff into [npm](https://npmjs.org/), and so I think perhaps today would be a good time to explain why I think this is the right way to go, and how you can use it in your own JS projects.

The big picture reason that all of this stuff is important, is that there a huge trend these days towards single page browser based applications and [JavaScript](http://en.wikipedia.org/wiki/JavaScript) development in general.  Many factors have contributed to this situation, like the rise of [fast JavaScript interpreters](http://en.wikipedia.org/wiki/V8_(JavaScript_engine)); widespread frustration and distrust of web plugins like [Silverlight](http://www.microsoft.com/silverlight/), [Java Applets](http://news.cnet.com/8301-1009_3-57563730-83/oracle-releases-software-update-to-fix-java-vulnerability/) and [Flash](http://techland.time.com/2011/11/09/mobile-flash-abandoned-for-html5-adobe-surrenders-apple-wins/); and the emergence of powerful new APIs like [HTML 5](http://www.w3.org/html/wg/drafts/html/master/) and [WebGL](http://www.chromeexperiments.com/webgl/).  Today, JavaScript runs on pretty much everything including desktops, cell phones, tablets and even game consoles.  Ignoring it is something you do at your own risk.

Module Systems

However, while there are many good reasons to use JavaScript, it has a reputation for scaling poorly with project size.  In my opinion, the main cause of this is that JavaScript lacks anything even remotely resembling a coherent [module system](http://en.wikipedia.org/wiki/Modular_programming).  This omission makes it inordinately difficult to apply sensible practices like:

* Hiding implementation details behind interfaces
* Splitting large projects into multiple files
* Reusing functionality from libraries and other code bases

Ignoring these problems isn’t an option.  But despite its flaws, JavaScript deserves at least some credit for being a very flexible language, and with some ingenuity one can work around the lack of modules.  There are of course many ways you can do this, and for me at least as I’ve been learning JS I’ve found the bewildering array of solutions pretty confusing.  So in this post I’m going to summarize how I understand the situation right now, and give my assessment of the various options:

Ad-Hoc

The obvious way to emulate a module in JavaScript would be to use a closure.  There is a fancy name for this in the JS community, and it is called the [module design pattern](http://www.adequatelygood.com/2010/3/JavaScript-Module-Pattern-In-Depth):

var MyModule = (function() {

var exports = {};

//Export foo to the outside world

exports.foo = function() {

// ...

}

//Keep bar private

var bar = { /\* ... \*/ };

//Expose interface to outside world

return exports;

})();

Here MyModule would become the interface for the module and the implementation of the module would be hidden away within the function() block.  This approach, when combined with features like [“use strict”;](https://developer.mozilla.org/en-US/docs/JavaScript/Reference/Functions_and_function_scope/Strict_mode) can go a long way to mitigating the danger of the global-namespace free-for-all; and if applied with discipline it basically solves the first problem on our big list.

Unfortunately, it doesn’t really do much to fix our multiple files or importing problems.  Without bringing in third party tools or more complex module loaders, you are basically limited to two ways to do this within a browser:

* You can add each of your JS files using a <script> in your header.
* Or you can just concatenate all your files in a static build process.

The latter approach is generally more efficient since it requires fewer HTTP requests to load and there are tools like [Google’s closure compiler](https://developers.google.com/closure/compiler/) which support this process.  The former approach is more dynamic since you can develop and test without having to do a full rebuild.  Unfortunately, neither method lets you to do selective imports, and they don’t handle external dependencies across libraries very well.

This style of coding is sometimes called monolithic, since the pressure to avoid code dependencies tends to push projects written in this fashion towards monumental proportions.  As an example, look at popular libraries like [jQuery](http://jquery.com/), [YUI](http://yuilibrary.com/) or [THREE.js](https://github.com/mrdoob/three.js/), each of which is packed with loads of features.  Now I don’t mean to especially pick on any of these projects, since if you aren’t going to accept external tooling their approach is actually quite reasonable.  Packaging extra functionality into a library (in the absence of a proper importing system) means you get a lot more features per <script>.  Sufficiently popular libraries (like jQuery) can leverage this using their own content distribution networks and can be cached across multiple sits.  This is great for load times, since users then don’t have to re-download all those libraries that their browser already has cached when they go from site-to-site.

The cost though is pretty obvious.  Because <script> tags have no mechanism to handle dependencies, the default way to include some other library in your project is [copy-paste](http://shtylman.com/post/the-copy-and-paste-virus/).  Heavy use of this style is a great way to proliferate bugs, since if you include a library with your project via this mechanism, you usually have no good way to automatically update it.

CommonJS

The [CommonJS module system](http://www.commonjs.org/specs/modules/1.0/) solves all these problems.   The way it works is that instead of running your JavaScript code from a global scope, CommonJS starts out each of your JavaScript files in their own unique module context (just like wrapping it in a closure).  In this scope, it adds two new variables which you can use to import and export other modules: **module.exports** and **require**.  The first of these can be used to expose variables to other libraries.  For example, here is how to create a library that exports the variable “foo”:

//In library.js

exports.foo = function() {

//... do stuff

}

And you can import it in another module using the “require” function:

var lib = require("./library.js");

lib.foo();

The addition of this functionality effectively solves the module problem in JavaScript, in what I would consider the most minimally invasive way.  Today, CommonJS is probably the most widely adopted module system in JavaScript, in part due to the support from [node.js](http://nodejs.org/).  The other major factor which has contributed to its success is the centralized [npm package manager](https://npmjs.org/), which makes it easy to download, install and configure libraries written using CommonJS.

CommonJS packages should be lean and mean, with as little extraneous functionality as possible.  Compare for example popular modules like [async](https://github.com/caolan/async) or [request](https://github.com/mikeal/request) to monolithic libraries like [underscore.js](http://underscorejs.org/) or [jQuery](http://jquery.com/).  An interesting discussion of this philosophy can be found at James Halliday (substack)’s blog:

J. Halliday, “[the node.js aesthetic](http://substack.net/posts/b96642/the-node-js-aesthetic)” (2012)

Based on the success of node.js, this approach seems to be paying off for writing server side applications.  But what about browsers?

CommonJS in the Browser

It turns out that doing this directly isn’t too hard if you are willing to bring in an extra build process.  The most tool to do this is [browserify](https://github.com/substack/node-browserify).  When you run browserify, it crawls all your source code starting from some fixed entry point and packages it up into a single .js file, which you can then minify afterwords (for example, using [uglify.js](https://github.com/mishoo/UglifyJS)).  Doing this reduces the number of http requests required to start up a page and reduces the overall size of your code, thus improving page load times.  The way it works is pretty automatic, just take your main script file and do:

browserify index.js > bundle.js

And then in your web page, you just add a single <script> tag:

<script src="bundle.js"></script>

browserify is compatible with npm and implements most of the [node.js standard library](http://nodejs.org/api/index.html) (with the exception of some obviously impossible operating system specific stuff, like process.spawn).

Rapid Development with browserify

The above solution is pretty good when you are looking to package up and distribute your program, but what if you just want to code and debug stuff yourself with minimal hassle? You can automate the bulk of this process using a Makefile, but one could argue reasonably that having to re-run make every time you edit something is an unnecessary distraction.  Fortunately, this is a trivial thing to automate, and so I threw together a tiny ~100-line script that runs browserify in a loop.  You can get it here:

[serverify – Continuous development with browserify](https://github.com/mikolalysenko/Serverify)

To use it, you can just run the file from your project’s root directory and it will server up static HTML on port 8080 from ./www and bundle up your project starting at ./index.js. This behavior can of course be changed by command line options or configuration files.  As a result, you get both the advantages of a proper module system and the ease of continuous deployment for when you are testing stuff!

Other Formats

As it stands today, CommonJS/npm (especially when combined with tools like browserify) is an efficient and  comprehensive solution to the module problem in JavaScript.  In my opinion, it is definitely the best option if you are starting out from scratch.  But I’d be remiss if I didn’t at least mention some of the other stuff that’s out there, and how the situation may change over time:

* Looking ahead to the distant future, [ECMA Script 6](http://wiki.ecmascript.org/doku.php) has a proposal for some new module semantics that seems promising.  Unfortunately, we aren’t there yet, and no one really knows exactly what the final spec will look and how far out it will be until it gets there.  However, I hope that someday they will finally standardize all of this stuff and converge on a sane, language-level solution to the module problem.  For a balanced discussion of the relative merits of the current proposal,  I’d recommend reading the following post by Isaac Schlueter (current maintainer of node.js):

Schlueter, I. “[On ES6 Modules](http://blog.izs.me/post/25906678790/on-es-6-modules)” (2012)

* Another module format is [AMD, or Asynchronous Module Definition](https://github.com/amdjs/amdjs-api/wiki/AMD).  Originally created as part of the [Dojo web framework](http://dojotoolkit.org/), it has since out grown its creators and developed its own small ecosystem.   Compared to CommonJS, the main differences of AMD are:
  1. Special syntax for specifying module imports (must be done at the top of each script)
  2. No tooling required to use, works within browsers out of the box.

These choices are motivated by a preceived need to get modules to work within browsers with no additional tooling.  The standard reference implementation of AMD is the [RequireJS](http://requirejs.org/) library, and you can install it on your sever easily by just copy/pasting the script into your wwwroot.

The fact that AMD uses no tools is both its greatest strength and weakness.  On the one hand, you can use it in a browser right away without having to install anything. On the other hand, AMD is often much slower than CommonJS, since you have to do many async http requests to load all your scripts.  A pretty good critique of the AMD format can be found here:

Dale, T. “[AMD is not the answer](http://tomdale.net/2012/01/amd-is-not-the-answer/)“

I generally agree with his points, though I would also like to add that a bigger problem is that AMD does not have a robust package manager.  As a result, the code written for AMD is scattered across the web and relies on copy-paste code reuse.  There are [some efforts under way](https://github.com/component) to fix this, but they are nowhere near as advanced as npm.

Conclusion

I’ve now given you my summary of the state of modules in JavaScript as I understand it, and hopefully explained my rationale for why I am picking CommonJS for the immediate future.  If you disagree with something I’ve said, have any better references or know of something important that I’ve missed, please leave a comment!