History and Background of JavaScript Module Loaders

Introduction

Application logic for web apps continues to move from the back end to the browser. But as rich client-side JavaScript apps get larger, they encounter challenges similar to those that old-school apps have faced for years: sharing code for reuse, while keeping the architecture separated into concerns, and flexible enough to be easily extended.

One solution to these challenges has been the development of JavaScript modules and module loader systems. This post will focus on comparing and contrasting the JavaScript module loading systems of the last 5-10 years.

It's a comprehensive subject, since it spans the intersection between development and deployment. Here's how we'll cover it:

1. A description of the problems that prompted module loader development
2. A quick recap on module definition formats

The problems

If you only have a few JavaScript modules, simply loading them via <script> tags in the page can be an excellent solution.

```
<head>

 <title>Wagon</title>

 <!-- cart requires axle -->

 <script src=“connectors/axle.js”></script>

 <script src=“frames/cart.js”></script>



 <!-- wagon-wheel depends on abstract-rolling-thing -->

 <script src=“rolling-things/abstract-rolling-thing.js”></sc

 <script src=“rolling-things/wheels/wagon-wheel.js”></script



```

Phone (optional)

Company/Organization *

How Many People Need Training? *

Just Me ▾

Message *

How can we help?

Submit

```
<!-- our-wagon-init hooks up completed wheels to axle -->

<script src="vehicles/wagon/our-wagon-init.js"></script>

</head>
```

However, <script> establishes a new http connection, and for small files – which is a goal of modularity – the time to set up the connection can take significantly longer than transferring the data. While the scripts are downloading, no content can be changed on the page (sometimes leading to the Flash Of Unstyled Content). Oh yeah, and until IE8/FF3 browsers had an arbitrary limit of 2 simultaneous downloads because bad people back in the day were bad.

The problem of download time can largely be solved by concatenating a group of simple modules into a single file, and minifying (aka uglifying) it.

```
<head>

<title>Wagon</title>

<script src="build/wagon-bundle.js"></script>

</head>
```

The performance comes at the expense of flexibility though. If your modules have inter-dependency, this lack of flexibility may be a showstopper. Imagine you add a new vehicle type:

```
<head>
```

```

<title>Skateboard</title>

<script src="connectors/axle.js"></script>

<script src="frames/board.js"></script>

<!-- skateboard-wheel and ball-bearing both depend on abstr

<script src="rolling-things/abstract-rolling-thing.js"></scr

<script src="rolling-things/wheels/skateboard-wheel.js"></sc

<!-- but if skateboard-wheel also depends on ball-bearing --

<!-- then having this script tag here could cause a problem

<script src="rolling-things/ball-bearing.js"></script>

<!-- connect wheels to axle and axle to frame -->

<script src="vehicles/skateboard/our-sk8bd-init.js"></script

</head>
```

Depending on the design of the initialization function in skateboard-wheel.js, this code could fail because the script tag for ball-bearing wasn't listed between abstract-wheel and skateboard-wheel. So managing script ordering for mid-size projects got tedious, and in a large enough project (50+ files), it became possible to have a dependency relationship for which there wasn't any possible order that would satisfy all the dependencies.

Modular programming

Modular programming, which we explored in a previous post, satisfies those management requirements nicely. Don't head out to celebrate just yet though – while we have a beautifully organized and decoupled codebase, we still have to deliver it to the user.

The stateless and asynchronous environment of the web favors user experience over programmer convenience. For example, users like it when they can start reading a web page before all of its images have finished downloading. But bootstrapping a modular program can't be so tolerant: a module's dependencies must be available before it can load. Since http won't guarantee how long that fetch will take, waiting for dependencies to become available gets tricky.

Javascript Module formats, and their loaders

The Asynchronous Module Definition (AMD) API arrived to solve the asynchronous problem. AMD takes advantage of the fact Javascript is processed in two phases: parsing (interpretation), when the code is checked for syntax, and execution, when the interpreted code is run.

amdjs-api
Created by amdjs
Star

Houses the Asynchronous Module Definition API groups.google.com/group/amd-implement

374 FORKS **3.0K** STARS

At syntax time, dependencies are simply declared in an array of strings. The module loader checks if it has that dependency loaded already, and performs a fetch if not. Only once all the dependencies are available (recursively including all dependencies' dependencies) does the loader execute the function portion of the payload, passing the now-initialized dependency objects as arguments to the payload function.

```
// myAMDModule.js

define([‘myDependencyStringName’, ‘jQuery’], function (myDep(
```

This technique solved the file ordering problem. You could again bundle all your module files into one big fella – in any order – and the loader would sort them all out. But it presented other advantages too: there suddenly became a network effect to using publicly hosted versions of common libraries like jQuery and Bootstrap. For example, if the user already had Google's CDN version of jQuery on their machine, then the loader wouldn't have to fetch it at all.

Ultimately, AMD's 'killer app' wasn't even a production feature. The best and least expected advantage of using a module loader came from referencing individual module files during development, and then seamlessly transitioning to a single, concatenated and minified file in production.

On the server side, http fetches are rare, as most files already exist on the local machine. The CommonJS format uses this assumption to drive an synchronous model. The CJS-compatible loader will make available a function named require(), which can be called from within any ordinary Javascript to load a module.

```
// myCommonJSModule.js

var myDepObj = require('myDependencyStringName');

var $ = require('jQuery');

if ($.version <= 1.6) alert('old JQ!');
```

CJS eliminates the cumbersome boilerplate syntax of AMD's define() signature. Back-end pros who use languages like Python often find CJS a more familiar pattern than AMD. CJS can also be statically analyzed more easily – for instance, in the example above, an analyzer could infer that the object being returned from require('jQuery') should have a property named 'version'. IDEs can use this analysis for useful features like refactoring and autocomplete.

Less Quick Start

Writing Effective JavaScript

Since require() is a blocking function, it causes the Javascript interpreter to pause the current code and switch execution context to require's target. In the example above, the console log won't execute until the code from myDependencyStringName.js has loaded and finished.

In the browser, downloading each dependency serially as the file is processed would result in even a small app having unacceptable load times. This doesn't mean no one can use CJS in the browser though. The trick comes from doing recursive analysis during build time – when the file has to get minified and concatenated anyway, the analyzer can traverse the Abstract Syntax Tree for all the dependencies and ensure everything gets bundled in the final file.

Finally, ES6, the most significant update to Javascript in many years, added built in support in the form of the new 'module' keyword. ES6 modules incorporate many of the lessons learned from both AMD and CJS, but resemble CJS more strongly, especially in regards to loading.

Reasons not to use a module loader

These days, modular programming and module loaders have become synonymous with rich web apps. But using modular programming does not necessarily require using a module loader. In my experience, only the issue of complicated module interdependence qualifies as absolutely requiring a module loader, but many projects have complicated loading infrastructure they just don't need.

Adding any technology to your stack has a cost: it increases both the number of things that can possibly go wrong and that you need to understand. Many of the benefits of loaders are just that – benefits – and not requirements. Beware the benefits that sound like no-brainers – You Ain't Gonna Need It, as it's a subtle form of premature optimization.

Try running your project without a loader at first. You'll have greater control over and insight into your code. If find you never need one, you're ahead, and adding one later is not hard.

The same YAGNI logic applies to the features of whatever module loader you choose. I've seen many projects use AMD named modules for no benefit whatsoever (and there's a substantial cost to it as well). KISS.

Tiny loaders

Early on, as AMD emerged as a leading client-side format, the module loading ecosystem exploded to support it. Libraries from this explosion include LAB.js, curljs, and Almond. Each had a different approach, but much in common: they were tiny (1-4kb), and followed the Unix philosophy of doing one thing and doing it well.

amdjs-api
Created by amdjs
Star

Houses the Asynchronous Module Definition API groups.google.com/group/amd-implement

374 FORKS 3.0K STARS

almond JavaScript
Created by requirejs
Star

A minimal AMD API implementation for use after optimized builds

161 FORKS 2.3K STARS

curl JavaScript
Created by cujojs
Star

curl.js is small, fast, extensible module loader that handles AMD, CommonJS Modules/1.1, CSS, HTML/text, and legacy scripts. github.com/cujojs/curl/wiki

187 FORKS 1.7K STARS

The thing they did was to load files, in order, and afterwards call back a provided function. Here's an example from the LABjs github:

```
<script src="LAB.js"></script>

<script>

$LAB
```

```

.script("http://remote.tld/jquery.js").wait()

.script("/local/plugin1.jquery.js")

.script("/local/plugin2.jquery.js").wait()

.script("/local/init.js").wait(function(){

initMyPage();

});

</script>

```

In this example, LAB starts fetching jQuery, waiting until it finishes executing to load plugin1 and plugin2, and waiting until those are finished before loading init.js. Finally, when init.js finishes, the callback function invokes initMyPage.

All these loaders use the same technical mechanism to fetch content: they write a <script> tag into the page's DOM with the src attribute filled in dynamically. When the script fires an onReadyStateChange event, the loader knows the content is ready to execute.

LAB and curl aren't actively maintained anymore, but they were so simple they probably still work in today's browsers. Almond still gets maintained as the minimalistic version of Require.

RequireJS

Require appeared in 2009, a latecomer among the tiny loaders, but went on to gain the greatest traction due to its advanced features.

requirejs JavaScript

Created by requirejs

Star

A file and module loader for JavaScript
requirejs.org

1.8K FORKS **9.7K** STARS

At its core, Require is not fundamentally different than the tiny loaders. It writes script tags to the DOM, listens for the finishing event, and then recursively loads dependencies from the result. What made Require different was its extensive – some might say baffling – set of configuration options and operating sugar. For example, there are two documented ways to kick off the loading process: either pointing an attribute named data-main of the script tag that loads RequireJS at an init file…

```
<script src="tools/require.js" data-main="myAppInit.js" ></

...or invoking a function named require() in an inline scrip

<script src="tools/require.js"></script>

<script>

require(['myAppInit', 'libs/jQuery'], function (myApp, $) {

</script>
```

...but the documentation recommends not using both, without giving a reason. Later, it's revealed the reason is neither data-main nor require() guarantee that require.config will have finished before they execute. At this point, inline require calls are further recommended to be nested inside a configuration call:

```
<script src="tools/require.js"></script>

<script>

require(['scripts/config'], function() {

require(['myAppInit', 'libs/jQuery'], function (myApp, $) {

});

</script>
```

Require is a swiss army knife of configuration options, but an air of automagical uncertainty hangs over the multitude of ways in which they affect each other. For example, if the baseUrl config option is set, it provides a prefix for the location to search for files. This is sensible, but if no baseUrl is specified, then the default value will be the location of the HTML page that loads require.js – unless you used data-main, in which case *that* path becomes baseUrl! Maps, shims, paths, and path fallback configs provide more opportunities to solve complex problems while simultaneously introducing unrelated ones.

Worth mentioning is possibly the most “gotcha” of its conventions, the concept of “module ID”. Following a Node convention, Require *expects* you to leave the ‘.js’ extension off the dependency declaration. If Require sees a module ID that ends in ‘.js’, or starts with a slash or an http protocol, it switches out of module ID mode and treats the string value as a literal path.

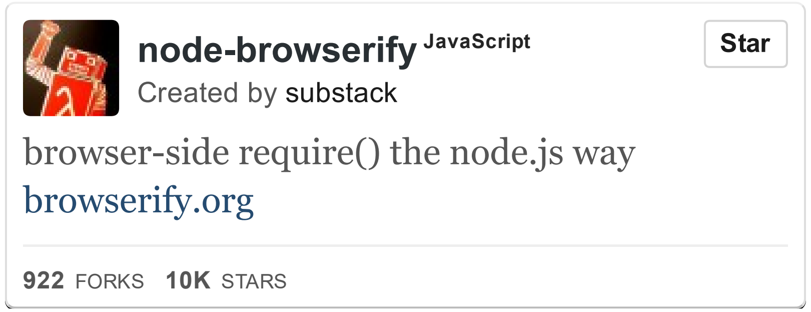
If we changed our example above like so:

```
require([‘myAppInit.js’, ‘libs/jQuery’], function (myApp, $)
```

Require is almost certain to fail to find myAppInit, unless it happens to be in the directory the baseUrl/data-main algorithm returns. As close to muscle memory as typing the ‘.js’ extension is, this error can be annoying until you get in the habit of avoiding it.

Despite all its idiosyncrasy, the power and flexibility of Require won it wide support, and it’s still one of the most popular loaders on the front end today.

Browserify



node-browserify JavaScript
Created by substack

Star

browser-side require() the node.js way
browserify.org

922 FORKS 10K STARS

Browserify set out to allow use of CommonJS formatted modules in the browser. Consequently, Browserify isn't as much a module loader as a module bundler: Browserify is entirely a build-time tool, producing a bundle of code which can then be loaded client-side.

Start with a build machine that has node & npm installed, and get the package:

npm install -g –save-dev browserify

Write your modules in CommonJS format, and when happy, issue the command to bundle:

browserify entry-point.js -o bundle-name.js

Browserify recursively finds all dependencies of entry-point and assembles them into a single file:

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

Adapted from server-side patterns, Browserify does demand some changes in approach. With AMD, you might minify and concat “core” code, and then allow optional modules to be loaded a la carte. With Browserify, all modules have to be bundled; but specifying an entry point allows bundles to be organized based on related chunks of functionality, which makes sense both for bandwidth concerns and modular programming.

Launched in 2011, Browserify is going strong.

Webpack

Webpack follows Browserify’s lead as a module bundler, but adds enough functionality to replace your build system. Expanding beyond CJS, Webpack supports not only AMD and ES6 formats, but non-script assets such as stylesheets and HTML templates.

webpack JavaScript
Created by webpack

A bundler for javascript and friends. Packs many modules into a few bundled assets. Code Splitting allows to load parts for the application on demand. Through "loaders," modules can be CommonJs, AMD, ES6 modules, CSS, Images, JSON, Coffeescript, LESS, ... and your custom stuff.
webpack.github.io

1.8K FORKS 18K STARS

Webpack runs on a concept called ‘loaders’, which are plugins registered to handle a file type. For example, a loader can handle ES6 transpilation (Webpack 2.0 handles ES6 natively), or SCSS compilation.

Loaders feed data into a “chunk”, which starts from an entry point – conceptually similar to a Browserify bundle. Once Webpack is set up, chunks are regenerated automatically as assets change. This can be very powerful, as you don’t have to remember to edit chunks.

The feature that has everybody really excited is hot module replacement. Once Webpack is in charge of your chunks, while running webpack-dev-server, it knows enough to modify code in the browser as you change the source. While similar to other source watchers, webpack-dev-server doesn’t require a browser reload, so it falls into the category of productivity tools that shave milliseconds off your dev process.

Basic usage is beyond simple. Install Webpack like Browserify:

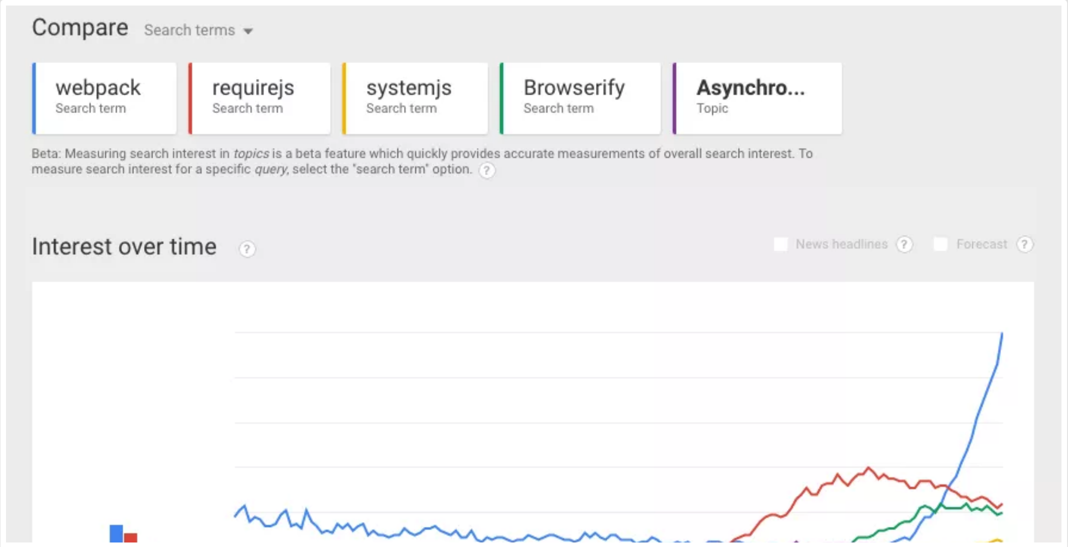
npm install -g –save-dev webpack

And pass the command an entry point and an output file:

webpack ./entry-point.js bundle-name.js

If you’re limiting use to Webpack’s impressive set of defaults, that command power always comes at a cost though. On one project, our team had several difficult problems – transpiled ES6 didn’t work after Webpack chunked it, and then SCSS worked locally but failed to compile in the cloud. In addition, Webpack’s loader plugin syntax overloads the argument to require(), so it won’t work outside of Webpack without modification (meaning you won’t be able to share code between client and server side).

Webpack has its sights set on the being next-generation compiler for the web, but maybe wait for the next version.



Google Trends' Take

Source

SystemJS

Wikipedia defines a polyfill as "additional code which provides facilities that are not built into a web browser", but the ES6 Module Loader Polyfill which SystemJS extends goes beyond the browser. An excellent example of how agnostic modern Javascript has become about the environment it runs in, the ES6 Module Loader Polyfill can also be used via npm in a Node environment.

systemjs JavaScript

Created by systemjs

Star

Universal dynamic module loader

455 FORKS **6.6K** STARS

SystemJS can be thought of as the browser interface to the ES6 Module Loader Polyfill. Its implementation is similar to RequireJS: include SystemJS on the page via a script tag, set options on a configuration object, and then call System.import() to load modules:

```
<script src="system.js"></script>

<script>

// set our baseURL reference path

System.config({

baseURL: '/app'

});

// loads /app/main.js

System.import('main.js');

</script>
```

SystemJS is the recommended loader of Angular 2, so it already has community support. Like Webpack, it supports non-JS file types with loader plugins. Like Require, SystemJS also ships with a simple tool, systemjs-builder, for bundling and optimizing your files.

However, the most powerful component associated with SystemJS is JSPM, or JavaScript Package Manager. Built on top of the ES6 Module Loader Polyfill, and npm, the Node package manager, JSPM promises to make isomorphic Javascript a reality. A full description of JSPM is beyond the scope of this article, but there's great documentation at jspm.io, and many how-to articles available.

Comparison Table

Loader Category	Local module format	Server files	Server module format	Loader code
Tiny loaders	Vanilla JS	Same structure as local files	Same format as local files	curl(entryPoint.js')

RequireJS	AMD	Concatenated and minified	AMD	requirejs('entryPoint.js', function (eP) {
				// startup code
				
				});

			CommonJS	

Browserify	CommonJS	Concatenated and minified	inside AMD wrapper	<script src="browserifyBundle.js"></script>
Webpack	AMD and/or CommonJs (mixed OK)	"Chunked" – Concat and minify into feature groups	Webpack proprietary wrapper	<script src="webpackChunk.js"></script>
SystemJS	Vanilla, AMD, CommonJS, or ES6	same as local	SystemJS proprietary wrapper	System.import('entryPoint.js') .then(function (eP) { // startup code });

Conclusion

Today's plethora of module loaders constitutes an embarrassment of riches compared to just a few years ago. Hopefully this post helped you understand why module loaders exists and how the major ones differ.

When choosing a module loader for your next project, be careful of falling prey to analysis paralysis. Try the simplest possible solution first: there's nothing wrong with skipping a loader entirely and sticking with plain old script tags. If you really do need a loader, RequireJS+Almond is a solid, performant, well supported choice. Browersify leads if you need CommonJS support. Only upgrade to a bleeding edge entry like SystemJS or Webpack if there's a problem you absolutely can't solve with one of the others. The documentation for these bleeding-edge systems is arguably still lacking. So use all the time you save by using a loader appropriate to your needs to deliver some cool features instead.

Share this:

12

Related

Handling the Challenge of Shared State With Ngrx/Store in Angular 2	Coordinating teams and responsive design	Why do ES6 Classes exist and why now?
August 25, 2016	March 28, 2015	June 1, 2016
In "AngularJS"	In "Responsive Web Design"	In "es6"

June 13, 2016

es6 javascript

by Elias Carlston in

es6, General, JavaScript

2 Comments appendTo 1 Login

Recommend 4 Share Sort by Best

Join the discussion…

Arnold Babasa • 6 days ago

Ah. This is just what I needed. Thanks!

• Reply • Share ›

Eloi Simard Quesnel • 3 months ago

I am surprised you never mention IIFE's!

• Reply • Share ›

ALSO ON **APPENDTO**

ES5 Objects vs. ES6 Maps – The differences and similarities

9 comments • 2 months ago•

Avat **Ryan Scheel** — JS Objects do have a way of testing for property existence (.has() on Maps): Either

What is the difference between state vs. props in React?

3 comments • 3 months ago•

Avat **sourire09** — brilliant! great article, step by step, for newbies in react :)

How Good C# Habits can Encourage Bad JavaScript

1 comment • 3 months ago•

Avat **yekit** — Nice post but the code snippet for this page is not working.

Handling the Challenge of Shared State With Ngrx/Store in Angular

2 comments • 7 days ago•

Avat **John Hopkins** — Many thanks. I also would like to see the resultant code on GitHub. Failed to load

Contribute Courses Blog Locations Teach For Us

Get In Touch