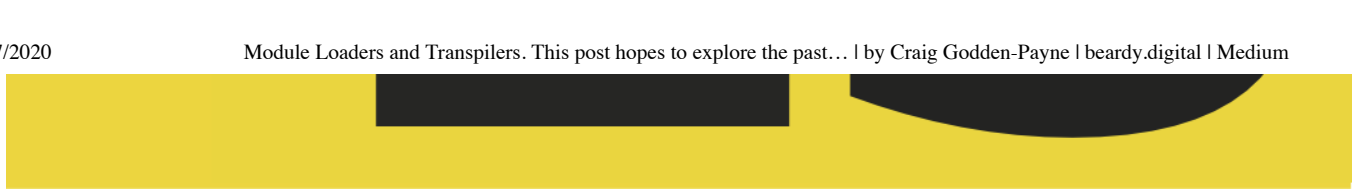
Module Loaders and Transpilers

Craig Godden-Payne
Oct 26, 2018 · 5 min read ★



This post hopes to explore the past, current and future state of build processes in front end technologies.

When working with node, its commonplace to use modules, but what I found there wasn't much information on how to do similar things in the browser, which is why I explored it a bit more. I wanted to understand the state of transpiling code, and module bundling, to allow current and older browsers to work with code written that uses modules or features that they can't support right now.

Why do we need to even do this?

We still live in a world, where no browser supports modules without us having to modify the code somehow. One of the first libraries to help tackle this problem was Browserify. Using CommonJS, it allowed modules to be used in the browser allowing all the required dependencies to be bundled into a single javascript file. You would typically a step in your build process (maybe gulp or grunt). Doing this allows us to benefit from encapsulation and dependency, something that never really existed in javascript in a true form. I have written a post a while back on the revealing module prototype pattern, which is interesting because when I wrote it, I was learning what now seems to be the precursor to modules in its current form.

We benefit from the following also:

- Use languages such as the latest version of ECMAScript, typescript or coffescript and feel safe that older browsers will support features.
- Use Npm, which contains over 475,000 shared packages, and they can very easily be used in the browser, as well as in node applications.

How would we load modules in the past?

In the past we would reference a script on the page, and rely on the load order. A good example of this is jQuery. When jQuery is loaded, it adds $ to the global scope or window. It also meant that if you had a library that depended on jQuery, you would have to make sure that the library was loaded afterwards, or typically you would do something on document loaded, to make sure that it would work. If you were adding

some of your own code, you would then tend to hide it under a namespace, such as window.MyNamespace, and hope that it wouldn't clash with anything else.

How will things be loaded in the future

We will be able to write code in languages such as ES2015 > export objects using a common module format, such as

```
export default class CraigsClass {
  Functionality (){
    alert("hello");
  }
}
```

We would then load this module, into another module using code such as

```
import CraigsClass from 'CraigsClass';
CraigsClass().Functionality();
```

The benefits of this are:

- The order of loading can be easily inferred from the code
- No namespaces are required on window
- Easy access to modules hosted in npm
- Easy access to other modules in your code

The Present

In its current state, running es2015 > code, with modules, in a browser means the code needs to be transpiled. After looking at the outputted code, it seems really complicated e.g.

```
"use strict";

Object.defineProperty(exports, "__esModule", {
  value: true
});
```

```
var _createClass = function () { function defineProperties(target,
props) { for (var i = 0; i < props.length; i++) { var descriptor =
props[i]; descriptor.enumerable = descriptor.enumerable || false;
descriptor.configurable = true; if ("value" in descriptor)
descriptor.writable = true; Object.defineProperty(target,
descriptor.key, descriptor); } } return function (Constructor,
protoProps, staticProps) { if (protoProps)
defineProperties(Constructor.prototype, protoProps); if
(staticProps) defineProperties(Constructor, staticProps); return
Constructor; }; }();

var _CraigsClass = require("CraigsClass");

var _CraigsClass2 = _interopRequireDefault(_CraigsClass);

function _interopRequireDefault(obj) { return obj && obj.__esModule
? obj : { default: obj }; }

function _classCallCheck(instance, Constructor) { if (!(instance
instanceof Constructor)) { throw new TypeError("Cannot call a class
as a function"); } }

(0, _CraigsClass2.default)().Functionality();

var CraigsClass = function () {
  function CraigsClass() {
    _classCallCheck(this, CraigsClass);
  }

  _createClass(CraigsClass, [{
    key: "Functionality",
    value: function Functionality() {
      alert("hello");
    }
  }]);

  return CraigsClass;
}();

exports.default = CraigsClass;
```

There are multiple module definitions such as commonjs, umd, amd

And you can bundle your assets using multiple different bundlers such as browserify, jspm, webpack and rollup, or you can define your own using gulp or grunt.

You can transpile your code using one of the many transpilers such as babel, coffeeScript or typescript.

And then you can load the modules dynamically using other libraries such as require.js or system.js

You can find a great article here The-cost-of-transpiling-es2015-in-2016 that compares various combinations of the above.

Frontend build processes

There are various build tools out there for building/transpiling/minifying/bundling frontend code. A few of the more popular ones are gulp and grunt, although you can use npm directly or work with a tool that can do it all, such as webpack.

What you will find is that the configuration for building frontend code seems very complicated, mostly because there are just so many different ways of doing it, and each way handles it differently! Theres a lot of very similar functionality that is provided by different libraries and a lot of wheel reinventing to try and create the next best build process. I don't think there is a clear best way to do it, but I guess everyone will have a faviourite after trying different tech.

More about modules

Modules are self-contained, and updating a module is easier if it is decoupled from other pieces of code. Modules can be reused, eliminating duplicate pieces of code thereby saving huge amount of time. In order to use modules currently, we need to use a script loader. Script loading is used so that we can use modular JavaScript in applications today, until it becomes part of modern browsers.

There are a number of loaders for handling module loading, and when you are building for production, its recommended to use optimization tools (like the RequireJS optimizer) to concatenate scripts.

AMD

The Asynchronous Module Definition (AMD) format is used to create a solution for loading javascript modules in the font end, that developers can use today, rather than waiting for browsers to support modules. The AMD module format allows the dependencies to asynchronously loaded. It removes the tight coupling between code and module identity. AMD uses the define method for facilitating module definition, and the require method for handling dependency loading. AMD was created as Commonjs isn't suited as well for the browsers. It makes for better startup times,

compared to commonjs, and these modules can be of different types such as objects, functions, constructors, strings, JSON, etc.

Written on March 10, 2018.

Originally published on https://craig.goddenpayne.co.uk/javascript-bundling/

JavaScript ES6 Amd Bundle