WHY AMD?

Module Purposes ..... § 1
The Web Today ..... § 2
CommonJS ..... § 3
AMD ..... § 4
Module Definition ..... § 5
Named Modules ..... § 6
Sugar ..... § 7
CommonJS Compatibility ..... § 8
AMD Used Today ..... § 9
What You Can Do ..... § 10

This page talks about the design forces and use of the Asynchronous Module Definition (AMD) API for JavaScript modules, the module API supported by RequireJS. There is a different page that talks about general approach to modules on the web.

MODULE PURPOSES

§ 1

What are JavaScript modules? What is their purpose?

- ***Definition**: how to encapsulate a piece of code into a useful unit, and how to register its capability/export a value for the module.*
- ***Dependency References**: how to refer to other units of code.*

THE WEB TODAY

§ 2

```
(function () {
    var $ = this.jQuery;

    this.myExample = function () {};
}());
```

How are pieces of JavaScript code defined today?

- *Defined via an immediately executed factory function.*
- *References to dependencies are done via global variable names that were loaded via an HTML script tag.*
- *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.*
- *It requires extra tooling to substitute a set of script tags into one tag for optimized deployment.*

This can be difficult to manage on large projects, particularly as scripts start to have many dependencies in a way that may overlap and nest. Hand-writing script tags is not very scalable, and it leaves out the capability to load scripts on demand.

COMMONJS

§ 3

```
var $ = require('jquery');
exports.myExample = function () {};
```

The original CommonJS (CJS) list participants decided to work out a module format that worked with today's JavaScript language, but was not necessarily bound to the limitations of the browser JS environment. The hope was to use some stop-gap measures in the browser and hopefully influence the browser makers to build solutions that would enable their module format to work better natively. The stop-gap measures:

- *Either use a server to translate CJS modules to something usable in the browser.*
- *Or use XMLHttpRequest (XHR) to load the text of modules and do text transforms/parsing in browser.*

The CJS module format only allowed one module per file, so a "transport format" would be used for bundling more than one module in a file for optimization/bundling purposes.

With this approach, the CommonJS group was able to work out dependency references and how to deal with circular dependencies, and how to get some properties about the current module. However, they did not fully embrace some things in the browser environment that cannot change but still affect module design:

- *network loading*
- *inherent asynchronicity*

It also meant they placed more of a burden on web developers to implement the format, and the stop-gap measures meant debugging was worse. eval-based debugging or debugging multiple files that are concatenated into one file have practical weaknesses. Those weaknesses may be addressed in browser tooling at some point in the future, but the end result: using CommonJS modules in the most common of JS environments, the browser, is non-optimal today.

AMD

§ 4

```
define(['jquery'], function ($) {
    return function () {};
});
```

The AMD format comes from wanting a module format that was better than today's "write a bunch of script tags with implicit dependencies that you have to manually order" and something that was easy to use directly in the browser. Something with good debugging characteristics that did not require server-specific tooling to get started. It grew out of Dojo's real world experience with using XHR+eval and wanting to avoid its weaknesses for the future.

It is an improvement over the web's current "globals and script tags" because:

- *Uses the CommonJS practice of string IDs for dependencies. Clear declaration of dependencies and avoids the use of globals.*
- *IDs can be mapped to different paths. This allows swapping out implementation. This is great for creating mocks for unit testing. For the above code sample, the code just expects something that implements the jQuery API and behavior. It does not have to be jQuery.*
- *Encapsulates the module definition. Gives you the tools to avoid polluting the global namespace.*
- *Clear path to defining the module value. Either use "return value;" or the CommonJS "exports" idiom, which can be useful for circular dependencies.*

It is an improvement over CommonJS modules because:

- *It works better in the browser, it has the least amount of gotchas. Other approaches have problems with debugging, cross-domain/CDN usage, file:// usage and the need for server-specific tooling.*
- *Defines a way to include multiple modules in one file. In CommonJS terms, the term for this is a "transport format", and that group has not agreed on a transport format.*
- *Allows setting a function as the return value. This is really useful for constructor functions. In CommonJS this is more awkward, always having to set a property on the exports object. Node supports module.exports = function () {}, but that is not part of a CommonJS spec.*

MODULE DEFINITION

§ 5

Using JavaScript functions for encapsulation has been documented as the module pattern:

```
(function () {
   this.myGlobal = function () {};
}());
```

That type of module relies on attaching properties to the global object to export the module value, and it is difficult to declare dependencies with this model. The dependencies are assumed to be immediately available when this function executes. This limits the loading strategies for the dependencies.

AMD addresses these issues by:

- *Register the factory function by calling define(), instead of immediately executing it.*
- *Pass dependencies as an array of string values, do not grab globals.*
- *Only execute the factory function once all the dependencies have been loaded and executed.*
- *Pass the dependent modules as arguments to the factory function.*

```
//Calling define with a dependency array and a factory function
define(['dep1', 'dep2'], function (dep1, dep2) {

    //Define the module value by returning a value.
    return function () {};
});
```

NAMED MODULES

§ 6

Notice that the above module does not declare a name for itself. This is what makes the module very portable. It allows a developer to place the module in a different path to give it a different ID/name. The AMD loader will give the module an ID based on how it is referenced by other scripts.

However, tools that combine multiple modules together for performance need a way to give names to each module in the optimized file. For that, AMD allows a string as the first argument to define():

```
//Calling define with module ID, dependency array, and factory function
define('myModule', ['dep1', 'dep2'], function (dep1, dep2) {

    //Define the module value by returning a value.
    return function () {};
});
```

You should avoid naming modules yourself, and only place one module in a file while developing. However, for tooling and performance, a module solution needs a way to identify modules in built resources.

SUGAR

§ 7

The above AMD example works in all browsers. However, there is a risk of mismatched dependency names with named function arguments, and it can start to look a bit strange if your module has many dependencies:

```
define([ "require", "jquery", "blade/object", "blade/fn", "rdapi",
         "oauth", "blade/jig", "blade/url", "dispatch", "accounts",
         "storage", "services", "widgets/AccountPanel", "widgets/TabButton",
         "widgets/AddAccount", "less", "osTheme", "jquery-ui-1.8.7.min",
         "jquery.textOverflow"],
function (require,    $,         object,          fn,          rdapi,
          oauth,    jig,          url,          dispatch,    accounts,
          storage,    services,    AccountPanel,           TabButton,
          AddAccount,           less,    osTheme) {

});
```

To make this easier, and to make it easy to do a simple wrapping around CommonJS modules, this form of define is supported, sometimes referred to as "simplified CommonJS wrapping":

```
define(function (require) {
    var dependency1 = require('dependency1'),
        dependency2 = require('dependency2');

    return function () {};
});
```

The AMD loader will parse out the require(") calls by using Function.prototype.toString(), then internally convert the above define call into this:

```
define(['require', 'dependency1', 'dependency2'], function (require) {
    var dependency1 = require('dependency1'),
        dependency2 = require('dependency2');

    return function () {};
});
```

This allows the loader to load dependency1 and dependency2 asynchronously, execute those dependencies, then execute this function.

Not all browsers give a usable Function.prototype.toString() results. As of October 2011, the PS 3 and older Opera Mobile browsers do not. Those browsers are more likely to need an optimized build of the modules for network/device limitations, so just do a build with an optimizer that knows how to convert these files to the normalized dependency array form, like the RequireJS optimizer.

Since the number of browsers that cannot support this toString() scanning is very small, it is safe to use this sugared form for all your modules, particularly if you like to line up the dependency names with the variables that will hold their module values.

COMMONJS COMPATIBILITY

§ 8

Even though this sugared form is referred to as the "simplified CommonJS wrapping", it is not 100% compatible with CommonJS modules. However, the cases that are not supported would likely break in the browser anyway, since they generally assume synchronous loading of dependencies.

Most CJS modules, around 95% based on my (thoroughly unscientific) personal experience, are perfectly compatible with the simplified CommonJS wrapping.

The modules that break are ones that do a dynamic calculation of a dependency, anything that does not use a string literal for the require() call, and anything that does not look like a declarative require() call. So things like this fail:

```
//BAD
var mod = require(someCondition ? 'a' : 'b');

//BAD
if (someCondition) {
    var a = require('a');
} else {
    var a = require('a1');
}
```

These cases are handled by the callback-require, `require([moduleName], function () {})` normally present in AMD loaders.

The AMD execution model is better aligned with how ECMAScript Harmony modules are being specified. The CommonJS modules that would not work in an AMD wrapper will also not work as a Harmony module. AMD's code execution behavior is more future compatible.

VERBOSITY VS. USEFULNESS

One of the criticisms of AMD, at least compared to CJS modules, is that it requires a level of indent and a function wrapping.

But here is the plain truth: the perceived extra typing and a level of indent to use AMD does not matter. Here is where your time goes when coding:

- *Thinking about the problem.*
- *Reading code.*

Your time coding is mostly spent thinking, not typing. While fewer words are generally preferable, there is a limit to that approach paying off, and the extra typing in AMD is not that much more.

Most web developers use a function wrapper anyway, to avoid polluting the page with globals. Seeing a function wrapped around functionality is a very common sight and does not add to the reading cost of a module.

There are also hidden costs with the CommonJS format:

- *the tooling dependency cost*
- *edge cases that break in browsers, like cross-domain access*
- *worse debugging, a cost that continues to add up over time*

AMD modules require less tooling, there are fewer edge case issues, and better debugging support.

What is important: being able to actually share code with others. AMD is the lowest energy pathway to that goal.

Having a working, easy to debug module system that works in today's browsers means getting real world experience in making the best module system for JavaScript in the future.

AMD and its related APIs, have helped show the following for any future JS module system:

- ***Returning a function as the module value**, particularly a constructor function, leads to better API design. Node has module.exports to allow this, but being able to use "return function (){}" is much cleaner. It means not having to get a handle on "module" to do module.exports, and it is a clearer code expression.*
- ***Dynamic code loading** (done in AMD systems via require([], function (){})) is a basic requirement. CJS talked about it, had some proposals, but it was not fully embraced. Node does not have any support for this need, instead relying on the synchronous behavior of require('), which is not portable to the web.*
- ***Loader plugins** are incredibly useful. It helps avoid the nested brace indenting common in callback-based programming.*
- ***Selectively mapping one module** to load from another location makes it easy to provide mock objects for testing.*
- *There should only be at most **one IO action for each module**, and it should be straightforward. Web browsers are not tolerant of multiple IO lookups to find a module. This argues against the multiple path lookups that Node does now, and avoiding the use of a package.json "main" property. Just use module names that map easily to one location based on the project's location, using a reasonable default convention that does not require verbose configuration, but allow for simple configuration when needed.*
- *It is best if there is an **"opt-in" call** that can be done so that older JS code can participate in the new system.*

If a JS module system cannot deliver on the above features, it is at a significant disadvantage when compared to AMD and its related APIs around callback-require, loader plugins, and paths-based module IDs.

AMD USED TODAY

§ 9

As of mid-October 2011, AMD already has good adoption on the web:

- *jQuery 1.7*
- *Dojo 1.7*
- *EmbedJS*
- *Ender-associated modules like bonzo, qwery, bean and domready*
- *Used by Firebug 1.8+*
- *The simplified CommonJS wrapper can be used in Jetpack/Add-on SDK for Firefox*
- *Used for parts of sites on the BBC (observed by looking at the source, not an official recommendation of AMD/RequireJS)*

WHAT YOU CAN DO

§ 10

If you write applications:

- *Give an AMD loader a try. You have some choices:*
  - *RequireJS*
  - *curl*
  - *lsjs*
  - *Dojo 1.7+*
- *If you want to use AMD but still use the **load one script at the bottom of the HTML page** approach:*
  - *Use the RequireJS optimizer either in command line mode or as an HTTP service with the almond AMD shim.*

If you are a script/library author:

- *Optionally call define() if it is available. The nice thing is you can still code your library without relying on AMD, just participate if it is available. This allows consumers of your modules to:*
  - *avoid dumping global variables in the page*
  - *use more options for code loading, delayed loading*
  - *use existing AMD tooling to optimize their project*
  - *participate in a workable module system for JS in the browser today.*

If you write code loaders/engines/environments for JavaScript:

- *Implement the AMD API. There is a discussion list and compatibility tests. By implementing AMD, you will reduce multi-module system boilerplate and help prove out a workable JavaScript module system on the web. This can be fed back into the ECMAScript process to build better native module support.*
- *Also support callback-require and loader plugins. Loader plugins are a great way to reduce the nested callback syndrome that can be common in callback/async-style code.*

Latest Release: 2.3.2