Module Systems in Javascript

Relevant Links

- Node's modules[1]
- CommonJS specification[2]
- AMD specification[3]
- require.js docs on AMD[4]
- cujo.js tutorials[5]

Notes

Modules are the building blocks of applications in most modern programming languages.

- The module has an associated "name". Other modules can refer to it with that name.
- The module has a specific "interface", a certain functionality, that it provides. The rest of the application communicates with the module only through that interface. This provides a **separation of concerns**.
- Modules provide **encapsulation**: What happens within a module stays within the module. The module many contain many private functions and constants that perform various tasks, but only expose a small part.
- A module could be swapped with another module that provides the same interface, without affecting the rest of the application.
- In many object-oriented languages, the role of modules is often performed by classes.

In Javascript, there are at least 3 different kinds of approaches to modules. There are also "ES6 modules", that are part of the ECMAScript 6 specification, but we will not be discussing these here.

Manual Modules We can build a barebones "module" structure using globals and careful naming. This does not require any extra infrastructure.

CommonJS Modules This module style is used by Node and other mostly server-side technologies.

AMD Modules This is a paradigm that evolved to serve the asynchronous loading needs of client-side applications. It requires the use of "amd loaders".

[1] http://nodejs.org/api/modules.html
[2] http://wiki.commonjs.org/wiki/Modules/1.1
[3] https://github.com/amdjs/amdjs-api
[4] http://requirejs.org/docs/whyamd.html#amdtoday
[5] http://know.cujojs.com/tutorials

UMD Modules A "Universal Module Definition" has emerged as a collection of ways to merge 2 or more of the above types.

We will now discuss each of these cases in more detail.

Manual Modules

Manual modules offer a simple way to provide some namespacing capabilities. They use immediate function invocations to create a local scope with a specific export.

You typically start by using one global variable named after the author or application under consideration. That variable is an object whose properties hold the different modules for the application.

For example, in our application, called *PanthR*, we would create, if it does not already exist, a global variable called PanthR, and populate it with different modules. For instance we need a module called Variable to represent the statistical notion of a variable. Here's how such a file might look like:

```
// Create global PanthR if it doesn't already exist
var PanthR = PanthR || {};
PanthR.Variable = (function() {
    // Local scope that only methods related to Variable see
    var Variable;
    // ... module code here
    // ... you can refer to other modules:
    PanthR.OtherModule.doSomething();
    return Variable;
}());
```

This can work well for moderately sized projects. We can combine all these files in one big file, and the immediate function invocations keep the different scopes separate. Or we can put them one at a time in their own <script> tags (though one big file tends to be more efficient to download).

This is a simple format, and requires some discipline on the part of the programmer, but nothing special otherwise.

Its main drawback is that it offers no way to specify the dependencies between modules. For instance, when we access PanthR.OtherModule in the code above, how to do we know that it has already been created? We don't, we must rely on making sure we load/concatenate the files in their proper order. And this is something we must ourselves keep track of, a very fragile process.

CommonJS Modules

The CommonJS Module format was created by a group interested in using Javascript technologies on the server, like for instance Node.js (but there are others). Any such technology must provide certain libraries for input and output, managing the file system, multiple processes etc.

In the CommonJS module specification there are 3 provides globals:

- require is a function that takes as argument the module name returns the object exported by that module.
- exports the object exported by the module. The module can provide functionality by adding properties to this object.
- module is an object containing properties describing the module. In particular it contains an module.id property that is a string that can be used with require to load the module, and a module.exports property, which is the exported module. In fact exports is a variable initially set to the object in module.exports. So one often sets module.exports to the desired return object/function, rather than adding methods to exports.

Each file in the CommonJS specification is assumed to have its own local environment (as opposed to files loaded via <script> tags in the browser, where they are all treated as part of the global environment).

For the rest of the discussion, and examples, we will focus on the Node.js take on the specification, which has some minor variations.

But briefly here is how code would typically look like in a Node module:

```
// We read the "filesystem" module
var fs = require("fs");
// Load the "os" module
var os = require("os");
// Load some other custom modules of our own:
var otherModule = require("./otherModule");

// Implement our module
var myModule = {
    ...
};

// Ensure that myModule is what is exported
module.exports = myModule;
// No need to explicitly return anything
```

The string passed to the "require" call is used to locate the file. It is *resolved* to a full path to a file via a set of rules:

- Paths starting with "." or ".." are computed relative to the current file.
- The ".js" extension is to be omitted.
- If the path is a system-provided library (like "fs" or "os" or any of the other parts of the API[6]) then it is resolved in that fashion.
- If it is not a system-provided library, and it does not start with "." or "..", then it is resolved relative to a "node_modules" folder. It typically will start with the node_modules folder at the root of your project, and will later look at system-specified locations. You can find the details in the node modules page[7].

The required modules are loaded synchronously. This is an important characteristic of this module format.

[6] http://nodejs.org/api/
[7] http://nodejs.org/api/modules.html

AMD Modules

AMD stands for *Asynchronous Module Definition*. It is a specification[8] born out of a need to have modular development in a project that is meant to be deployed in the browser. It consists of a number of parts:

AMD Modules You write your module files in a specific format, with the use of the define function that we will discuss shortly. Part of that specification is what other modules your module depends on.

AMD Modules cannot be inserted into a webpage directly. They need the use of a loader or builder.

The key features of these modules is that they have explicitly declared dependencies, and that they can be loaded asynchronously: You specify the file's dependencies, along with how your module will finish its loading once those dependencies have been loaded.

AMD Loader The loader is responsible for loading the modules in the correct order. It has to provide a "define" function, and it processes the information provided in those "define" calls to determine the correct order in which modules should be loaded to resolve the dependencies.

There are a number of existing loaders, for including require.js[9] and RaveJS[10]. We will spend more time looking closer at require.js in future segments.

AMD Builder There are various programs whose goal is to build/consolidate the various AMD modules into one file to be served in a <script> tag. Loaders are used in the development of the application, while builders are used in the deployment phase. require.js[11] includes such a builder/optimizer, but there are many others, for instance browserify[12].

These programs often also do compression, removing spaces, comments etc, to reduce the file size and make it faster to download.

The key component of and AMD module is the define function. It takes up to three arguments:

- id the first argument, and it is optional. It is a string characterizing the module's "identifier". This often simply defaults to the filename, and is typically omitted.
- dependencies is the second argument, also optional but usually included. It is an *array* of the "id"s of modules that your module depends on. Those modules will be processed first before your module is processed. There are three special id

[8] https://github.com/amdjs/amdjs-api/blob/master/AMD.html
[9] http://requirejs.org/
[10] https://github.com/RaveJS
[11] http://requirejs.org/
[12] http://browserify.org/

names that are treated separately: "require", "exports" and "module". If those ids appear, they are resolved to their CommonJS module meaning.

If this second argument is omitted, then it defaults to the triple ["require", "exports", "module"].

One key difference with the ids in the AMD specification is that they are looked for either relative to the current file or from "top-level". There is nothing analogous to the "node_modules" folders.

- factory is the third, and only required, argument. It can be an object, in which case it is what is exported by this module. Or more typically it is a function, which will be executed exactly once, and its return value is the exported object from the module. This function will receive as arguments the modules that were listed in the dependencies array.
- For a proper AMD implementation, the define function has a property, define.amd, which must be an object but has no other required fields.

Here is an example of how such a file might look like. In its simplest form it has no dependencies:

```
define(function() {
    var OurModule;
    // ... define OurModule here
    return OurModule;
});
```

Here's an example of a module that depends on jQuery and one other module:

```
define(["jquery", "otherModule"], function($, otherM) {
    // "$" here will equal the jquery object
    // and otherM equal the "otherModule"
    // ...
    return MyModule;
});
```

We can easily transform a CommonJS module into a Node module by using the simplified CommonJS wrapping:

```
define(function(require, exports, module) {
    var a = require('a'),
        b = require('b');

    exports.action = function () {};
    // Or ...
    module.exports = MyModule;
});
```

A sample of using AMD modules can be found here[13]. Start by looking inside the js/app folder and the bottom of the index-test.html file for bootstrapping such an application. You will find there the following line:

```
<script type="text/javascript" data-main="js/app/main" src="js/lib/require.js"></script>
```

This line loads the require.js file, which then takes care of loading the application by starting at the "js/app/main" module and following its dependencies.

[13] https://github.com/skiadas/HealCalc3/tree/master/