UMD Modules

Relevant Links

- UMD repository[1]

Notes

In this section we look at a collection of formats for "Universal Module Definitions". These are meant to be files that can function in multiple environments simultaneously.

The goal of these patterns is to offer compatibility between AMD loaders, Node.js, and browser globals.

Before we move on, let's recall the main module types and their differences:

globals
- immediate function invocation creates a scope.
- other modules are accessed as properties of the global object
- module is exported as property of the global object
- modules loaded in order of script load in the html file

CommonJS
- each file assumed to run on its own scope
- other modules accessed via require
- module is exported by setting module.exports
- module are loaded synchronously

AMD
- whole file is a define call with input a *factory function* function(require, exports, module).
- other modules are accessed via require or as arguments to the factory function
- module is exported as the return value of the factory function
- modules are loaded asynchronously

Without much ado, here are some examples.

AMD and browser global

See also here[2].

This example provides a module if "define" is present, and creates a browser global otherwise.

```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD. Register as an anonymous module.
        define(['otherModule'], factory);
    } else {
```

[1] https://github.com/umdjs/umd
[2] https://github.com/umdjs/umd/blob/master/templates/amdWeb.js

```
        // Browser globals
        root.ourModule = factory(root.otherModule);
    }
}(this, function (otherModule) {
    // This is the factory function that is what is usually seen
    // in AMD code.

    // All your code goes here

    return ourModule;
}));
```

Let us walk through the code, as it takes some getting used to. The code consists of an immediate function invocation. That function receives as inputs the root object and the factory function that is meant to create the object.

What this function does is examine if define exists and is a function and has a define.amd property. If so, then we are in the AMD setting, and therefore want to perform a "define" call. Otherwise, we are meant to use browser globals, so store under "root" the result of calling the factory function, passing it the values of the other modules.

We then call this function, providing it this as a first argument (which is set to the global object) and the factory function as the second argument. All the actual module code goes here.

This requires us to do some self-maintainance, especially regarding the names of the modules, which need to be kept in multiple places. One could attempt to automate the process a bit, but that is probably more trouble than it's worth.

AMD, Node and Browser global

See also here[3]. Here is a version that attempts to also work in Node:

```
(function (root, factory) {
    if (typeof define === 'function' && define.amd) {
        // AMD. Register as an anonymous module.
        define(['otherModule'], factory);
    } else if (typeof exports === 'object') {
        // Node.
        module.exports = factory(require('otherModule'));
    } else {
        // Browser globals
        root.ourModule = factory(root.otherModule);
    }
}(this, function (otherModule) {
    // This is the factory function that is what is usually seen
    // in AMD code.

    // All your code goes here

    return ourModule;
}));
```

[3] https://github.com/umdjs/umd/blob/master/templates/returnExports.js

One of the problems you will encounter is the way in which 'otherModule' is looked at in the Node version. You would need to use relative paths to make sure it works properly, or else you might have to alter the path slightly.

AMD with Node Adapter

See also here[4].

This format feels somewhat better than the previous formats, but doesn't play well with browser globals. It effectively uses the "simple CommonJS wrapping" format for AMD, where the dependencies are loaded via require calls, but it provides a simple define function to accomodate Node.

As an intermediate step, consider that instead of the normal AMD style where we have a define directly, we can do the following:

```
// Instead of:
define(function (require, exports, module) {
    ...
});
//
// We do:
(function(define) {
    define(function (require, exports, module) {
        ...
    });
}(define));
// --^^^^-- Immediate function invocation binds the global define
// to the parameter define
```

With that in mind, the following code simply feeds on the external function a custom-made define, if need be:

```
(function(define) {

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

        // Your code goes here

        return function () {};
    });

}( // Help Node out by setting up define.
     typeof module === 'object' && typeof define !== 'function'
    ? function (factory) { module.exports = factory(require, exports, module); }
    : define
));
```

This is an immediate function invocation, and the function is provided a define. If we are in AMD then there is a define already, so we pass it to the function. Otherwise we assume that we are in a Node setting and define a barebones define function via: function (factory) { module.exports = factory(require, exports, module); }.

[4] https://github.com/umdjs/umd/blob/master/templates/nodeAdapter.js

In other words, we provide a define function that expects to be given a factory function, which in turn expects arguments require, exports and module. The AMD define would provide those arguments (as it defaults to a dependencies array of ["require", "exports", "module"]). In the absence of AMD but presence of Node, which has those 3 defined directly, we simply call the factory with them. The returned value is meant to be the exported module, so we assign it to module.exports.