Module Systems in Javascript

Relevant Links

- ES6 modules[1]
- Node's modules[2]
- CommonJS specification[3]

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 may 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 these days there are at least 3 different kinds of approaches to modules.

Manual Modules We can build a barebones "module" structure using globals and careful naming. This does not require any extra infrastructure, but it is limited. You are also responsible for making sure dependencies are loaded in the correct order.

Node Modules This module style is used by Node and other mostly server-side technologies. It uses two special methods, require and exports, one for including other modules and the other for exporting functionality to the rest of the world.

ES6 Modules This type of module was introduces with ECMAScript 6, and they are fully supported by all modern browsers. It uses the keywords import and export

AMD Modules This is a paradigm that evolved to serve the asynchronous loading needs of client-side applications, and it requires the use of "amd loaders". It is less useful today in the presence of ES6 modules.

[1] http://exploringjs.com/es6/ch_modules.html
[2] http://nodejs.org/api/modules.html
[3] http://wiki.commonjs.org/wiki/Modules/1.1

One of the challenges of Javascript is that these different approaches don't always play well with each other.

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.

The key elements of approach are as follows:

- We use an anonymous function which is immediately invoked to protect the insides of our application from the rest of the world. Anything we define within the function stays local to the function unless we make steps to make it available outside.
- We use a specific global variable object as a *namespace*, possibly named after our organization or project. Our different modules will become properties of that object.
- We access other modules through the global names they may have. This runs the risk of someone else overwriting our global name.

Here's how such a file might look like:

```
// Function takes as argument the "global" object. In browser that is "window".
// In Node, it is "global". This is provided at the bottom of the snippet,
// where the function is called immediately after its definition.
(function(root) {
    let MyOrg = root.MyOrg || {};
    root.MyOrg = MyOrg;
    let OtherModule = MyOrg.OtherModule; // Loading another module

    // Creating the module. Could use a class declaration or whatever is appropriate.
    let MyModule = ...;

    // We do stuff for our module.
    // We can use OtherModule here.

    // Store this module so other modules can find it.
    MyOrg.MyModule = MyModule;
}(typeof window === 'undefined' ? global : window));
```

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 MyOrg.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.

Node (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 provided globals:

- require is a function that takes as argument the module name and returns the object exported by that module.
- exports is 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 a 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
// Modules "paths" that don't start with a "./" are searched in
// the "module" space, a special library directory.
let fs = require("fs");
// Load some other custom modules of our own.
// This path is relative to where our current file is at.
let otherModule = require("./otherModule");

// Implement our module
let myModule = {
    ...
};
```

```
// Ensure that myModule is what is exported
module.exports = myModule;
// No need to explicitly return anything. The file can just end.
```

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[4]) then it is resolved as such.
- 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[5].

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

ES6 Modules

ES6 Modules are the current module specification for Javascript. It is already implemented in most browsers, and Node.js is in the process of creating support for it.

We often use the extension .mjs for ES6 modules to distinguish them from Node modules. And when we want to include them into a Javascript page, we must use the type="module" attribute of the script tag:

```
<script type="module" src="....."></script>
```

One of the ongoing challenges is that some of the current support for ES6 Modules is limited: When it comes to testing frameworks, we have to jump through a few hoops to make it work smoothly.

ES6 Modules follow use the following primitive constructions:

- import is used to load another module. You may choose to import everything that is exported from that module, or some selected pieces.
- export is used to specify which parts of the module are meant to be exported and used by other modules. You can export multiple functions/classes/objects, using export in front of each of them (or grouping them together), or you can use export default to export exactly one thing. In the former case, you must use the correct names for all the pieces you want to use; in the latter you can choose on the importing module what name you want to give to the imported module. export default is particularly suited to modules that are essentially classes.

[4] http://nodejs.org/api/
[5] http://nodejs.org/api/modules.html

Make sure to read these notes[6] for more details.

Here is a small example of what a ES6 module might look like.

```
// lib is a library that exports a "square" and a "diag"
import { square, diag } from 'lib';
square(...)
// Or ...
import * as lib from 'lib';
lib.square(...)
```

[6] http://exploringjs.com/es6/ch_modules.html