2. Node Module System

Global object

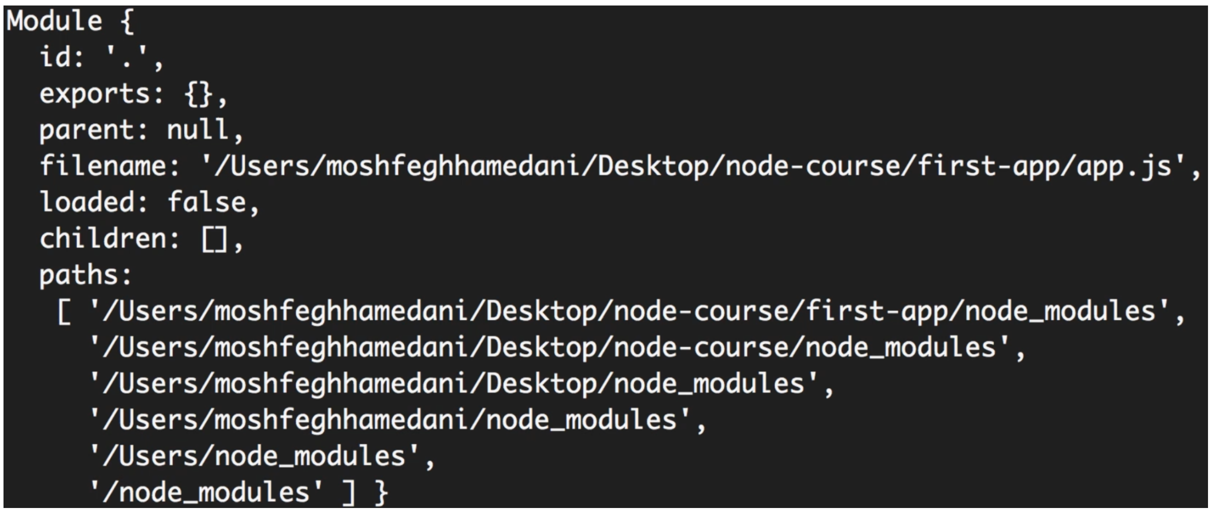
In JavaScript, there is always a global object defined. In a web browser, when scripts create global variables defined with the *var* keyword, they are created as members of the global object. (In Node.js this is not the case.) The global object's interface depends on the execution context in which the script is running. For example:

- In a web browser, any code which the script does not specifically start up as a background task has a Window as its global object. This is the vast majority of JavaScript code on the Web.
- Code running in a Worker has a WorkerGlobalScope object as its global object.
- Scripts running under Node.js have an object called *global* as their global object.

Modules

In a real world application, we often split out JS code into multiple files, so it is possible that we have two files with the same function. Because this function is added to the global scope, when we define this function in another file, that new definition is going to override the previous definition. In order to build reliable and maintainable applications we should avoid defining variables and functions in the global scope, instead we need modularity. We need to create small modules where we define our variables and functions, so two variables / functions with the same name will not be overridden.

Every file in a Node app is considered a Module. The variables and functions defined in that module are scoped in that file, in OOP they would be considered private. If you want to use them outside that module you need to explicitly export it and make it public. Every Node app has at least one Module called the *main Module*. If you do `console.log(module)` you can see the following:



```
Module {
  id: '.',
  exports: {},
  parent: null,
  filename: '/Users/moshfeghhamedani/Desktop/node-course/first-app/app.js',
  loaded: false,
  children: [],
  paths:
   [ '/Users/moshfeghhamedani/Desktop/node-course/first-app/node_modules',
     '/Users/moshfeghhamedani/Desktop/node-course/node_modules',
     '/Users/moshfeghhamedani/Desktop/node_modules',
     '/Users/moshfeghhamedani/node_modules',
     '/Users/node_modules',
     '/node_modules' ] }
```

Creating a Module

One of the properties of the Module object is *exports*, everything that we add to exports will be available outside the Module.
The *module.exports* is a special object which is included in every JavaScript file in the Node.js application by default. The *module* is a variable that represents the current Module, and *exports* is an object that will be exposed as a Module. So, whatever you assign to *module.exports* will be exposed as a Module.

Export Literals

As mentioned above, *exports* is an object. So it exposes whatever you assigned to it as a Module. For example, if you assign a string literal then it will expose that string literal as a Module.

The following example exposes a simple string message as a module in Message.js.

```
module.exports = 'Hello world';
```

Now, import this message module and use it as shown below.

```
var msg = require('./Messages.js');
console.log(msg);
```

You must specify ./ as a path to the root folder to import a local module. However, you do not need to specify the path to import Node.js core modules or NPM modules in the require() function.

Export Object

The *exports* is an object. So, you can attach properties or methods to it. The following example exposes an object with a string property in Message.js file.

```
exports.SimpleMessage = 'Hello world';
//or
module.exports.SimpleMessage = 'Hello world';
```

In the above example, we have attached a property SimpleMessage to the exports object. Now, import and use this module, as shown below.

```
var msg = require('./Messages.js');
console.log(msg.SimpleMessage);
```

In the above example, the require() function will return an object `{ SimpleMessage : 'Hello World'}` and assign it to the msg variable. So, now you can use msg.SimpleMessage.

In the same way as above, you can expose an object with function. The following example exposes an object with the log function as a Module.

```
module.exports.log = function (msg) {
    console.log(msg);
};
```

The above module will expose an object `{ log : function(msg){ console.log(msg); } }`. Use the above Module as shown below.

```
var msg = require('./Log.js');
msg.log('Hello World');
```

You can also attach an object to *module.exports*, as shown below.

```
data.js
module.exports = {
    firstName: 'James',
    lastName: 'Bond'
}

app.js
var person = require('./data.js');
console.log(person.firstName + ' ' + person.lastName);
```

Export Function

You can attach an anonymous function to *exports* object as shown below.

```
module.exports = function (msg) {
    console.log(msg);
};
```

Now, you can use the above Module, as shown below.

```
var msg = require('./Log.js');
msg('Hello World');
```

The *msg* variable becomes a function expression in the above example. So, you can invoke the function using parenthesis *( )*.

Export Function as a Class

In JavaScript, a function can be treated like a class. The following example exposes a function that can be used like a class.

```
module.exports = function (firstName, lastName) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.fullName = function () {
        return this.firstName + ' ' + this.lastName;
    }
}
```

The above Module can be used, as shown below.

```
var person = require('./Person.js');
var person1 = new person('James', 'Bond');
console.log(person1.fullName());
```

As you can see, we have created a person object using the new keyword.

Loading a Module

To load a Module we use the *require()* function. This is one of the functions that we do not have in browsers. This function takes one argument and that is the name or path of the target Module that we want to lead. When the .js extension is not loaded, Node assumes that it is a JS Module and it adds it automatically. For example:

```
require('./logger');
```

In the recent versions of JS we have the ability to define constants, so as a best practice when loading a Module using the *require* function it is better to store the result in a constant. The reason for this is because we do not want to accidentally override the value of the Module that we are importing.

```
const logger = require('./logger');
```

Module Wrapper Function

Node does not run our code directly, it wraps the entire code inside a function before execution. This function is termed as Module Wrapper Function.

https://nodejs.org/api/modules.html#modules_the_module_wrapper

Before a module's code is executed, Node wraps it with a function wrapper that has the following structure:

```
(function (exports, require, module, __filename, __dirname) {
  //module code
});
```

The top-level variables declared with *var*, *const*, or *let* are scoped to the Module rather than to the global object. It provides some global-looking variables that are specific to the module, such as:

- The *module* and *exports* objects that can be used to export values from the module.
- The variables like __filename and __dirname, that tells us the module's absolute filename and its directory path.