**Learning Node**

**Modules – part 2**

Modules are essential to the way that Node puts together functionality. Node has both user defined modules (either local modules, which you defined, or third party-modules), and core modules.

Our key topics here are:

1. **An overview of modules**
2. **Where do I find and how do I choose modules?**
3. **How do I install modules?**
4. **How do modules provide functionality to other parts of my application (i.e. *export*  functionality and *require(some\_module)*  )?**
5. **How are dependencies of one file on a module documented (i.e. *package.json*)**
6. **Exactly what is export**ed **? and some simple examples  
     
   In part 1 of modules we discussed items 1-5. We now move on to 6, and some simple examples.**

**Review – Installing a module**

We can install a module as part of a (local) project from the project folder with:  
   
 **npm install *my\_module***

If we have already created a project with  **npm init** and have a package.json file, then we add my\_module to the dependencies with:

**npm install my\_module –save  
  
 NOTE: You must access npm outside node – i.e. you must first stop node (clt+c in Windows)**If you want the module you install to be available globally you type  
  
 **npm install *my\_module* –g** (The –g flag may go either before or after the name of the module)  
  
Usually global installs are recommended only for modules whose functions will be used on the command line.  
  
  
<http://www.tutorialspoint.com/nodejs/nodejs_npm.htm> states that “Globally installed packages/dependencies are stored in system directory. Such dependencies   
can be used in CLI (Command Line Interface) function of any node.js but can not be imported using require() in Node application directly. “ but my  
experience is tht you can use require() after a global install.

**Review: How the module exposes some functions or values and how your app.js makes use of what is exported.**a. Typically a module will export an object:  
  
 module.export = {  
 func1: func1,  
 func2: func2,  
 const1: const1  
 }  
 where func1 and func2 are previously defined functions and const1 is a value.  
  
 Once in a while a module will export only one function (or value):  
  
 module.export = soloFunc;  
  
b. To make use of a module globalModule which was installed *globally* (such as a core module) inside app.js (or other js file) you code:  
  
 **const global\_module = require('global\_module');**  
  
To make use of a module someModule which you installed locally (inside your project folder) inside app.js (or other js file) you code:  
  
 **const some\_module = require('./some\_module');**  
  
c. Either way, if the module exported an object then you use the usual dot notation to access the functions/values in the object. For example, if the someModule was installed locally , required as above, and someModule exported func1 and func2 then we could write:

let func1 = some\_module.func1;

If someModule exported just soloFunc, then soloFunc was assigned to someModule when we did the require.

1. **How are dependencies of one file on a module documented (i.e. *package.json*)   
   a. Basic answer**: When you are in the folder with your main application (traditionally named app.js) you type  
     
    **npm init**This will generate a series of questions and after you have answered them it will generate a **package.json** file.  
   The package.json file will document these answers, including dependencies (see numbers 3 and 4 above.)  
     
   **REFERENCES**<http://freecontent.manning.com/managing-node-js-project-dependencies-with-npm-and-a-package-json-file/> on managing dependencies. Clear, straighforward.  
   [http://www.infoworld.com/article/2984358/application-development/inside-npm-building-and-sharing-javascript- packages.html?phint=newt%3Dinfoworld\_daily&phint=idg\_eid%3Dae8e3ea0bca8a3c279ed6a8235932ec2#tk.IFWNLE\_nlt\_daily\_pm\_2015-09-22](http://www.infoworld.com/article/2984358/application-development/inside-npm-building-and-sharing-javascript-%20%20packages.html?phint=newt%3Dinfoworld_daily&phint=idg_eid%3Dae8e3ea0bca8a3c279ed6a8235932ec2#tk.IFWNLE_nlt_daily_pm_2015-09-22) on what npm will do for you, but less detailed.  
   <https://www.packtpub.com/books/content/using-nodejs-dependencies-nwjs?mc_cid=23ec57dbd9&mc_eid=9aa1af28be> on what modules are shipped with node

**b. Details about versions of modules**These can be found either in the freecontent link above or in Chapter 3 of the Ali Syed book (see references or class by class assignments) Beginning Node.  
  
If at some point you want to write your own module, and add it to the Node.js ecosystem, then the API for modules is at <https://nodejs.org/api/module.html> and for general documentation about modules <https://nodejs.org/api/modules.html>   
  
There are 3 types of modules- core, local, and third-party.

* **Core modules** provide the most important functions, and they are automatically loaded whenever Node starts up. A clear description of them is at <https://www.tutorialsteacher.com/nodejs/nodejs-modules> and they are summarized in:

| Core Module | Description |
| --- | --- |
| [http](https://nodejs.org/api/http.html) | http module includes classes, methods and events to create Node.js http server. |
| [url](https://nodejs.org/api/url.html) | url module includes methods for URL resolution and parsing. |
| [querystring](https://nodejs.org/api/querystring.html) | querystring module includes methods to deal with query string. |
| [path](https://nodejs.org/api/path.html) | path module includes methods to deal with file paths. |
| [fs](https://nodejs.org/api/fs.html) | fs module includes classes, methods, and events to work with file I/O. |
| [util](https://nodejs.org/api/util.html) | util module includes utility functions useful for programmers. |

* **Local Modules** are the ones you write yourself
* **Third party modules** are the ones, other than the Core ones, which you will download from npm. They include many wonderful modules for creating templates for web pages, etc.

**c. Modules and versions**  
 Node.js uses a common system for numbering versions. The versions are numbered as X.Y.Z The Z indicates bug fixes (i.e. each version starts at X.Y.0 and then becomes X.Y.1 etc.) The Y indicates minor updates and the X indicates major updates. When X is *odd* that version is supported for 6 months. When X is *even,* that is an LTS (Long Term Support) version --- and that is what you want.  
 So when you start out you want the highest even number version.  
 As you know, backwards compatibility is always an issue. The folks who run Node.js know this and they have a slightly atypical solution to this. You can have different versions of Node and of Node modules running on different projects. That way, you don't need to worry about a new version killing your project.   
 On the other hand, it makes life a little more complex than with ES6+, where you need only one version, knowing that the ECMAScript people will always maintian backwards compatibility. For example, in Node you will always need to show the dependencies on specific versions of modules.   
 If you want more information about Node versions, go to <https://tamalweb.com/which-nodejs-version>   
 *Modules For Node-Part 1 has a discussion of CommonJS modules (what we will use) vs. ECMAScript Modules.*

1. **Exactly what is export**ed **? and some simple examples**
   1. When you require a module the functions in the module get wrapped (& turned into a *function* *expression*). One advantage of that is that the variables and methods which are local to your function are protected. On the other hand, thanks to closures, the object which is exported has access to all those variables and methods (b/c they were part of the context in which it was created.)  
      .
   2. If you require a module, then require looks for a .js file (unless you put on another extension).   
        
      If require can't find a .js file with that name then require will look for a folder with that name, and inside that folder it will look for a file named index.js
   3. It is possible to build multiple .js files, each of which exports a function, and then in the index.js file you can require those modules, assigning their functions to variables.  (Although I think I'd rather build the functions directly in index.js).
   4. Finally, your module.exports is an object which names those functions.

In the code below I am using red for variable (which may be a function) and blue for property names.

Make a folder named **Example1** and inside that folder make another folder named **greet.**The greet folder will hold a module we are writing and the Example1 folder will hold our app.js and the Greet folder.

Let’s put some files in our **greet folder** :

And the **greet folder** will have the following 3 files (english.js , spanish.js and index.js) :

**//english.js**

   var greet = function() {

console.log('Hello');

}

module.exports = greet;

**//index.js**

var english = require('./english');

var spanish = require('./spanish');

module.exports = {

english: english,

spanish: spanish

};

Now, inside greet ask for **npm init** --- ***it should choose index.js as the entry point*** - and then **npm install.** (The node modules will be empty.)

Finally, inside the app.js file you will require greet (the folder) and what will be returned from greet is what index.js exports ---- namely the json object.

So you can write for **app.js**

var greet =  require('./greet');  //looks for the index.js file in the greet folder

                                           //gets what module.exports says in index.js

greet.english();    //runs the function associated with the english property of greet

greet.spanish();   //runs the function associated with the spanish property of greet

The index.js file is grabbing functions from other modules (this could also be useful to grab stuff from core modules), and assigning them to variables.

Then the index files goes on to write some functions - could be using the stuff it grabbed - and assigns those functions to variables.

Finally, it exports (more than one of) them in a json object.

Now in your example1 folder type **npm init** This time the entry point will be app.  
Using the directories on your hard drive, if you look in the example1 and greet folders you should see the package.json files. Once you type npm install in these folders you will also get the package-lock.json files (which give a complete description of the dependency trees so that you can reproduce them with the exact same versions of various modules.) **7. Now on to exports vs. module.exports:**  
In the pages which follow you will find several writers’ explanations of the difference between ***exports*** and ***module.exports****.*The bottom line is that ***it is always module.exports which is returned and you can never go wrong by making sure that you build that – as a function or as an object or whatever else you need – and leaving exports alone (to point to module.exports).***

Experiment: Make a folder Example2. It should be just like Example 1, except that in the index file, change the code  
  
module.exports = {

english: english,

spanish: spanish

};  
  
to the code:

        module.exports.english  = english;

        module.exports.spanish = spanish;

Here's something else you might want to try as a different variant on Example1: change the index.js file so that exports uses anonymous functions – and note that it will no longer require the english.js and the spanish.js files. Call that Example3.

module.exports = {

   english:function() {console.log('Hello')},

   spanish:function() {console.log('Hola')}

}

Do we need to wrap those anonymous functions - e.g.

           :function() {console.log('Hello')}

inside parens to turn them into function expressions?  why or why not?

**Finally, do the following:**

Create a .json file named **greetings.js**

The contents of that file is

{

"en": "Hello",

"es": "Hola"

}

btw please notice that in json the **keys** need to be in quotes when they are strings.

Now change the **english.js** file to:

var greetings = require('./greetings.json');  //NOTE the .json extension must be there

var greet = function() {

console.log(greetings.en);   //just the en property of greetings

}

module.exports = greet;

and likewise for the **spanish.js**file.

Reference for above:  
<https://www.sitepoint.com/understanding-module-exports-exports-node-js/>

**7’ Here is a restatement of this same information from another author:**

I. Modules encapsulate code - either for reuse-ability or to make your code cleaner and easier to test.

II. When you write a module you may choose to make some of its functions publicly available.  You do this thru the **exports**method.

III. A typical structure might be to have app.js as your main js file and to have my\_mod.js

in the same folder with the helper functions.

my\_mod.js might have several functions and look like this:

func1 = function( ) {

    //do something

}

func2 = function( ) {

    //do something

}

func3 = function( ) {

     //do something

}

var exports = module.exports = {func1: func1,  
 func3:func3}

I believe you can also do the following in my\_mod.js:

func2 = function () {

    //do something

};

var exports = module.exports = {}

exports.func1 = function() {

   //do something

 };

exports.func3 = function() {

   //do something

};

Equivalently, in the above we could have said - see item VI.

module.exports.func1 = func1;

module.exports.func3 = function () {.....};

Now you need to get the stuff from my\_mod into app.js and this is done with the **require** method.

So in app.js you would say (sorry  about the convention in reusing names):

var my\_mod = require("./my\_mod.js");      // This is actually the path to my\_mod since it is in the same folder.

The require function will return whatever exports has exposed.

Note that you can actually say require("./my\_mod")   that is w/o the .js and node will assume it is a .js file.

And now you can call:

my\_mod.func1();

my\_mod.func3();

**7’’ Yet another point of view of the same issues:**

  Per  <http://openmymind.net/2012/2/3/Node-Require-and-Exports/>

There's another way to expose things in a module:

var User **=** **function**(name, email) {  
 **this**.name **=** name;  
 **this**.email **=** email;  
};  
  
module.exports **=** User;

The difference is subtle but important. See it? We are exporting user directly, without any indirection. The difference between:

module.exports.User **=** User;  
*//vs*module.exports **=** User; //I think this is subtle but not especially insightful.

If you prefer module.exports.User = User then you are exporting an object which has (at least one) key-value pair, namely the User key has as its value the User function.   
On the other hand if you prefer module.exports = User then you have killed the object that exports would typically use and what you exporting is just the User function.  
Obviously, then, when you require the user module you need to know what you are getting --- is it an object with a User: User pair, or is it a function which you can run directly.

is all about how it's used:

**var** user **=** require('./user');  
**var** u **=** **new** user.User();  
*//vs***var** u **=** **new** user();

Finally, the last thing to consider is what happens when you directly export a function:

var powerLevel **=** **function**(level) {  
 **return** level **>** **9000** ? "it's over 9000!!!" : level;  
 };

module.exports **=** powerLevel;

When you require the above file, the returned value is the actual function. This means that you can do:

require('./powerlevel')(**9050**);

Which is really just a condensed version of:

**var** powerLevel **=** require('./powerlevel')

powerLevel(**9050**);

**My comment: Be careful to distinguish between exporting a function and exporting what one call to that function returns!**

7”’. **module.exports vs. exports – a few more explanations**  
 a. From examples at <http://www.hacksparrow.com/node-js-exports-vs-module-exports.html>

**This page explains that if you leave module.exports unassigned, then all the properties which you assign to exports will be assigned to module.exports.**

**On the other hand, if you assign something (e.g. a JSON objet) to module.exports then that is what gets exported and you can't get a hold of the functions which you may have previously attached as properties to export.**

The advice here is "So you get the point now - if you want your module to be of a specific object type, use module.exports; if you want your module to be a typical module instance, use exports."

My comment: exports is a short cut synonym for module.exports and it *points to*  the object module.exports There is a lot of discussion of this (see section c below). I think we don't need to worry about it. (*What it is saying is that no matter what you set exports to, it is module.exports which is returned. They start out pointing to the same object, but you can break that connection by reassigning exports.)*

What you do need to be clear about is if you exporting an object, some of whose values may be functions, or if you are exporting just one function.  
It is always safer to go the object route. That way, if you later add some other functions to the exports the folks who used your original module won't have their code broken. Does this make sense to you?  
In the user module above, if I chose module.exports.User = User (or equivalently   
module.exports = {User:User}, which is the same) then anyone who requires the user module will make use of it with  
 const user = require('./user');  
 myUser = new user.User(theName, theEmail);

Now I the folks who maintain user decide to add a new method sendAds() what is exported may be module.exports = {User: User, advertise:sendAds }  
The two lines above (in the app which required the user module) will still work, even if it never makes use of the sendAds method.

On the other hand, if the user module had exported just the User function, then it couldn't also export the sendAds method without resorting to exportin and object & breakin everyone's code.

This page aslo has examples of specific object types - arrays, constructor functions, etc.

b. More patterns may be found at   
 <https://darrenderidder.github.io/talks/ModulePatterns/#/>

c. still more on **module.exports vs exports** See <http://stackoverflow.com/questions/7137397/module-exports-vs-exports-in-node-js> and its links.  
 Documentation at <https://nodejs.org/docs/latest/api/modules.html#module.exports> and <https://nodejs.org/docs/latest/api/modules.html#modules_module_exports> As noted here, exports is a variable and like any other variable when you make a new assignment you have over-written the previous value.   
   
 Clear explanation at <http://www.hacksparrow.com/node-js-exports-vs-module-exports.html>   
 As noted in the previous paragraph, *exports* is a variable. Of course, like anything else in JavaScript this variable is an *object*, so you may assign properties to it.  
 If, inside your module, your code says  
  
 exports = *something*;  
  
 then *something* is the one and only thing which will be exported.  
  
 On the other hand, if, inside your module, your code says  
  
 exports.property1 = *something*;  
 exports.property2 = *somethingElse*:  
  
 now exports will carry both those objects when the module is   
 exported.  
 **I need to try this and make sure it isn’t module.exports.property1 etc.**  
 More typically, follow the module.exports = function() example in the link above. As   
 hacksparrow summarizes:

…Your modules don't always have to be 'module instances'. Your modules can be any legal JavaScript object - boolean, number, date, JSON, string, function, array, and so on. Your module is whatever you set module.exports to. If you don't set module.exports to anything explicitly, the properties of exports and attached to it and returned….. So you get the point now - if you want your module to be of a specific object type, use module.exports; if you want your module to be a typical module instance, use exports.  
  
 At stackoverflow the same discussion has the following summaries:

**It's all about references. Think of exports like a local variable object pointing to module.exports. If you overwrite the value of exports, then you lose the reference to module.exports, and module.exports is what you expose as a public interface.**   
  
**Quick Summary:** both exports and module.exports point to the same object, unless you reassign one. And in the end module.exports is returned. So if you reassigned exports to a function then don’t expect a function since it isn't going to be returned. However if you had assigned function with code such as  exports.func = function... then the resulting thing would have func property with function as a value. Because you added the property to the object that exports was pointing to  
  
**It is module.exports that is returned!**