Module 3 Lab 1: Understanding Node.js Module System with Practical Examples

Ryan M.K. Burke-Jimenez

Arizona State University

IFT 458

Professor Dinesh Sthapit

September 24th, 2023

Module 3 Lab 1: Understanding Node.js Module System with Practical Examples

There are various aspects that are associated with the Node.js Module System. To put it simply, “Modules are the blocks of encapsulated code that communicate with an external application on the basis of their related functionality” (GeeksforGeeks, 2023). In this lab, we will demonstrate and explain the topics of Scope, wrapping/injection, and exporting Modules that are created within the lab environment.

**Scope**

Scope is a typical attribute within many programming languages that works to limit the availability (or “scope”) of a certain code or code snippet. This means that a line or section of code may be omitted or reserved for the rest of the application given the specific scope it has applied to it. There are several scopes we will be observing:

* Module Scope
* Block Level
* Local Level
* Closure Scope

**Module Scope.** The Module Scope allows the rest of the application to reach the code implemented without any issue (Figure 1). This scope is usually defined by writing in the body of the application that is not surrounded by brackets ([ ]), braces ({ }), or any other operator to define it as a function.

A screen shot of a computer

Description automatically generated

Figure - Module Scope

**Block Level.** The Block Level scope is defined by a “block” structure which is defined using {} braces. With the braces present, the scope allows only that block section to access the code supplemented within it. Figure 2 demonstrates that an error will be thrown when attempting to call for variable blockLevelVariable outside of the block. This shows precisely how the Block Level functions; once the call outside of the block is commented out, we can see a successful run and print to console with no error.

A screenshot of a computer

Description automatically generated

Figure - Block Level Scope

**Local Level.** The scope for a Local Level is like the Block Level, however, the coding snippet is within a function. The code that is used within the function can only be located by other parts within the function. As seen in Figure 3, the variable localLevelVariable is initialized within the function displayVar() and console.log() command will call the localLevelVariable successfully. However, when attempting to reach localLevelVariable outside of the function will not be successful and will print an error on the console. If the call for variable localLevelVariable outside of the function is commented out, there will be no error.

A screenshot of a computer program

Description automatically generated

Figure - Local Level Scope

**Closure Scope.** Closure Scope is an interesting scope that is established within functions. According to HubSpot writer Athena Ozanich, “Closure in JavaScript is a form of lexical scoping used to preserve variables from the outer scope of a function in the inner scope of a function” (2023). The way that I like to think of this scope is the idea of nesting functions. If there is a variable created in the outer function, it may be reached within the inner (nested) function, however, this logic is not inversed. Closure Scope is hierarchical in nature and only the outer functions can be reached by the inner but the outer cannot reach the inner function (Figure 4).

A screenshot of a computer program

Description automatically generated

Figure - Closure Scope

**Scope Reflection.** Scope is an important part for compartmentalizing code and keeping an organized structure as well as invoking hierarchy. This can be seen in concepts such as data abstraction, where only relevant data to the functionality of the application is presented upon execution of said application. With Scope, we can limit where data or instruction can travel, we can destinate code with structure, and we can reuse variable names at each Scope Level (although this may not be best practice).

**Understanding Node.js Module System**

In this section, we will be investigating more functions of Node.js. We will start by creating a quick mathModules.js file that includes basic mathematical functions (add, subtract, multiply, divide). Although functions are declared in this file, the main objective is to demonstrate module.exports in action (Figure 5).

*A screenshot of a computer

Description automatically generated*

Figure - Creating mathModules.js + module.exports

Although seemingly simple, the background functions of what this application is doing is quite complex. Module.exports() is a module that is handled within the Node.js framework but is established as follows:

(function(exports, require, module, \_\_filename, \_\_dirname){});

When module.exports() is called, the function is required, references the current module, as well as returning the path of the current module with \_\_filename and \_\_dirname.

**Working with Module.exports and Require**

The final portion of this lab is demonstrating the usage of module.exports and require in the Node.js framework. When exporting the custom module to your environment, it is imperative that the require() module is used within the application that you are using the exported module in. Require() is defined in the global scope, meaning, that you never have to require(‘require’). According to Samer Buna, writer for FreeCodeCamp, states the following functionality of the require() module (2017):

* Resolving – Sets an absolute path of the file that is required.
* Loading – Determines the type of file content.
* Wrapping – To give the file its own scope, allowing required objects to be locally accepted in all applications in the environment.
* Caching – Stores information in a easily accessible location to make call recursion simple.

In a new file named exportExample.js, we have examples of export and require being used to run simple arithmetic (Figure 6). We can see how the export and require modules work hand in hand, where the export in the call in the application that has the custom module allows the module to be locally found, and the require module will catch and allow the custom module to be present where the call is located.

A screenshot of a computer

Description automatically generated

Figure - Export and Require Showcase

**References**

GeeksforGeeks. (2023). Node.js modules. *GeeksforGeeks*. <https://www.geeksforgeeks.org/node-js-modules/#:~:text=41%20Likes-,In%20Node.,collection%20of%20multiple%20files%2Ffolders>.

Ozanich, A. (2023, June 21). JavaScript Closure: What is it & How do you use it? *HubSpot*. <https://blog.hubspot.com/website/javascript-closure#:~:text=What%20is%20closure%20in%20JavaScript,position%20in%20the%20source%20code>.

Buna, S. (2020). Requiring modules in Node.js: Everything you need to know. *freeCodeCamp.org*. <https://www.freecodecamp.org/news/requiring-modules-in-node-js-everything-you-need-to-know-e7fbd119be8/>