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

Saurabh Kale

IFT 458/554: Middleware Programming & Database Security

Dinesh Sthapit

Sep 24th, 2023

1.1. **Module Level Scope-**

A screenshot of a computer

Description automatically generated

Explanation-

The variable **moduleLevelVariable** is declared using the **let** keyword at the top level of a module or script, outside of any specific function or block. This placement makes it a "module level" variable, meaning it is accessible and can be used throughout the entire module. In this case, the variable is assigned the value **"Module Level"**. The **displayVar** function defined below it can access and log the value of **moduleLevelVariable** because it's within the same module, showcasing the variable's availability at the module level. When the **displayVar** function is called, it prints "Module Level" to the console, demonstrating that module-level variables are accessible within the module's scope.

1.2. **Block Level Scope-**

A computer screen shot of a black screen

Description automatically generated

Explanation-

The variable **blockLevelVariable** is declared using the **let** keyword within a block of code enclosed by curly braces **{}**. This type of variable is referred to as a "block-level" variable. Block-level variables are only accessible within the block in which they are defined. In this case, **blockLevelVariable** is scoped to the block in which it's declared, which is delimited by the curly braces.

When you attempt to log **blockLevelVariable** outside of the block, as in the second **console.log** statement, you receive a **ReferenceError** because the variable is not defined in that scope. Block-level variables are limited to the block they are declared in and are not accessible outside of it. This is useful for creating isolated scopes and avoiding naming conflicts in larger scripts or programs.

Top of Form

1.3. **Local Level Scope**:

A computer screen with many colorful text

Description automatically generated

Explanation-

The variable **localLevelVariable** is declared using the **let** keyword within the **displayVar** function. This type of variable is referred to as a "local level" variable. Local level variables are scoped to the function or block in which they are defined. In this case, **localLevelVariable** is a local variable, and it is only accessible within the **displayVar** function's scope.

When attempted to log **localLevelVariable** outside of the **displayVar** function, as in the second **console.log** statement, we will receive a **ReferenceError** because the variable is not defined in that scope. Local level variables are limited to the function or block in which they are declared and are not accessible outside of it. This scoping behavior helps maintain data encapsulation and prevents naming conflicts between variables in different parts of your code.

1.4. **Closure Scope**:

A computer screen shot of a program

Description automatically generated

Explanation-

A closure is a function that has access to the variables from its containing (or "enclosing") function, even after the outer function has finished executing. In this case, **outerFunction** defines a variable called **closureVariable**, and it then returns an inner function (**innerFunction**).

When you invoke **outerFunction** and assign the result to **newFunction**, you essentially create a closure. This means that **innerFunction** "remembers" the environment in which it was created, including the **closureVariable** declared in **outerFunction**. So, when you later call **newFunction()**, it can still access and log the value of **closureVariable**, which is why it prints "Closure Scope" to the console.

This behavior is powerful because it allows you to create functions that encapsulate and preserve data from their parent scopes, providing a way to maintain private data or create functions with persistent state.

2. Understanding Node.js Module system-

2.1 mathModule.js-

let add = (x, y) => x + y;

let subtract = (x, y) => x - y;

let multiply = (x,y) => x\*y;

let divide = (x,y) => x/y;

module.exports = {add, subtract, multiply, divide}

A computer screen shot of a black screen

Description automatically generated

Explanation-

A set of four functions for basic arithmetic operations (addition, subtraction, multiplication, and division) is defined using arrow function syntax. Each function takes two parameters, **x** and **y**, and returns the result of the corresponding mathematical operation applied to these parameters.

Next, the **module.exports** statement is used to make these functions available for use in other parts of a Node.js module or application. The **{ add, subtract, multiply, divide }** object is exported, effectively allowing other modules to import and use these functions by referencing them as properties of the imported object.

This code follows the common practice in Node.js of encapsulating related functionality within a module and making it accessible to other parts of the application through the **module.exports** mechanism. This modular approach enhances code organization and reusability, as other modules can easily import and utilize these arithmetic functions in their own code.

2.2-

A computer screen shot of a black screen

Description automatically generated

Explanation-

In Node.js, every module is wrapped within a function to create a private scope for that module. This wrapping function is automatically generated by Node.js and serves as a way to encapsulate the module's code. Within this function, several objects are injected as parameters, namely **exports**, **require**, **module**, **\_\_filename**, and **\_\_dirname**.

1. **exports**: The **exports** object is initially an empty object that allows a module to define what should be accessible to other modules when they import it. By adding properties and methods to this object, a module can expose its functionality to other parts of the application.
2. **require**: The **require** function is used to import other modules into the current module. It takes a module's path as an argument and returns the **exports** object of the imported module, allowing the current module to access its functionality.
3. **module**: The **module** object represents the current module. It contains information about the module itself, such as its filename, and also provides a way to export values using **module.exports**. Developers can use **module.exports** to define what the module exposes to other modules.
4. **\_\_filename**: **\_\_filename** is a variable that holds the absolute path to the current module's file. It can be useful for tasks like file manipulation or determining the location of the current module.
5. **\_\_dirname**: **\_\_dirname** is a variable that holds the directory path of the current module's directory. It provides the directory context in which the module is located and can be used to construct absolute file paths relative to the module's directory.

2.3

A computer screen shot of a black screen

Description automatically generated

Explanation-

In the provided code snippet, the injection of **exports**, **require**, **module**, **\_\_filename**, and **\_\_dirname** as parameters within the wrapping function is of paramount importance for creating modular and maintainable Node.js modules.

Firstly, **module.exports** allows us to explicitly define what parts of the module should be accessible to other parts of the application. In this case, the **add** and **subtract** functions are made available for other modules to use when they import this module. This encapsulation helps in keeping the module's internal details hidden while providing a well-defined public interface.

Secondly, the **require** function enables the module to import other modules, allowing for code reuse and structuring applications as a collection of interconnected modules.

Lastly, **\_\_filename** and **\_\_dirname** provide essential information about the module's location and context, which can be crucial for tasks like resolving relative paths to other resources or determining the module's file and directory locations.

3

A screenshot of a computer

Description automatically generated

Explanation-

When you use the **require** function in Node.js to import a module, several behind-the-scenes processes occur. In the example you provided, you have a module named **mathModule.js** that exports a set of functions, and you're importing it in **usingmathModule.js**. Here's what happens:

1. **Module Resolution**: When **require('./mathModule')** is called, Node.js first needs to locate the **mathModule.js** file. It begins by searching for **mathModule.js** in the current directory. If it doesn't find it there, it will look in the built-in Node.js modules or in the **node\_modules** folder of your project.
2. **Module Loading**: Once Node.js finds the **mathModule.js** file, it reads the file and wraps its contents in a function that includes parameters like **exports**, **require**, **module**, **\_\_filename**, and **\_\_dirname**. This wrapping ensures that the module's variables and functions are encapsulated and don't interfere with the global scope.
3. **Execution**: The code inside **mathModule.js** is executed within the context of the wrapping function. In this case, it defines several functions (**add**, **subtract**, **multiply**, and **divide**) and assigns them to the **module.exports** object. This object is used to specify what parts of the module should be accessible when it's imported in other files.
4. **Exports**: When **require('./mathModule')** is called in **usingmathModule.js**, Node.js returns the **exports** object of **mathModule.js**. This object contains references to the functions you defined in **mathModule.js**.
5. **Accessing Exported Functions**: In **usingmathModule.js**, you assign the returned **exports** object to the variable **math**, and you can now use **math.add(3, 5)** and **math.subtract(5, 3)** to call the functions exported from **mathModule.js**.

Top of Form

4. Wrap up and Conclusion –

The Node.js module system is a fundamental aspect of Node.js that facilitates code organization, reusability, and maintainability in JavaScript applications. It consists of several key concepts and processes that work together seamlessly:

1. **Module Definition**: Node.js modules are encapsulated units of code, typically stored in separate files. These modules can contain functions, variables, and other code elements.
2. **module.exports**: Each module can explicitly specify what should be accessible to other modules using the **module.exports** object. This object allows modules to define their public interface, making specific functions or variables available for import.
3. **require Function**: The **require** function is used to import modules into other modules. It takes a module's path as an argument and returns the **exports** object of the imported module, allowing the importing module to access its functionality.
4. **Module Wrapping**: Behind the scenes, Node.js wraps the code within each module in a function that includes parameters like **exports**, **require**, **module**, **\_\_filename**, and **\_\_dirname**. This wrapping ensures that the module's variables and functions are encapsulated and don't interfere with the global scope.
5. **Module Resolution**: When a module is imported using **require**, Node.js searches for the module's file based on the provided path. It looks in the current directory, built-in Node.js modules, and the **node\_modules** folder, following a specific resolution algorithm.
6. **Encapsulation**: The encapsulation provided by the module system ensures that variables and functions within a module are isolated from the global scope, reducing naming conflicts and promoting code reliability.
7. **Code Reusability**: Modules can be easily reused in different parts of an application or shared across multiple projects, enhancing code reusability and modularity.
8. **Private Scope**: Variables and functions within a module are private by default, meaning they are not accessible from outside the module unless explicitly exported.
9. **Dependency Management**: The module system handles dependencies between modules, ensuring that modules are loaded and executed in the correct order, minimizing runtime errors.