Episode-5-Diving Into NodeJS GitHub Repo

Why We can not access one module variable outside the module ?

How Require Function works ?

- so here we go back in js the scope concept

```
function x () {
    const a = 10;
    function b () {
        console.log("b");
    }
}

console.log(a) //a is not defiend
```

- when we write js code which is go in nodjs and nodejs have v8 engine inside it
- v8 engine follow the ESCMAScript . and just like js we can not access the Any function’s variable and function method outside the function . untill the module.export
- same concept we have when we write require(”./xyz.js”)
- this require function takes the module code and wrap it in the function and then you have access in your module
- means when you call require
    - all the code in module are wrap inside a function
    - and that function is special function which we called IIFE
    -

```
- If require xyz in app.js  then
```

```
xyz.js

IIFE - immediately invoked function expression

(function(){

      //All code of module run inside here and then this wrap goes to app.js

}) ()
```

- before going to code in v8 its wrap in IIFE
- In this pattern, you create a function and then immediately invoke it. This is different from a normal function in JavaScript, which is defined and then called
  separately:
- iife keeps variable and function private. inside function or variable not interfere with the outside module code

```
function x() {}
x();
```

q- how are the variable and function are private in different module ?

q- how do you get access to module.export ?

- when we export the all code will wrap inside the iife
- let take example of sum.js goes app.js

```
function(){
      require("./path")

      function calculateSum(a, b) {
         const sum = a + b;
         console.log(sum);
         }
```

```
    module.exports = { calculateSum };
}
```

- suppose the sum.js have also some require method then it also in this IIFE

Q- How does module and require in the function we get

- because nodejs make iife that time nodejs pass the parameter as “module” & “require()”

```
function(module,require,.....){ //there are more parameter too

    require("./path")

    function calculateSum(a, b) {
       const sum = a + b;
       console.log(sum);
       }

    module.exports = { calculateSum };
}
```

How require() Works Behind the Scenes

1 Resolving the Module
Node.js first determines the path of the module. It checks whether the path is a local file (./local), a JSON file (.json), or a module from the node_modules directory, among other possibilities.

2.Loading the Module
Once the path is resolved, Node.js loads the file content based on its type. The loading process varies depending on whether the file is JavaScript,JSON, or another type.

---

3 Wrapping Inside an IIFE
The module code is wrapped in an Immediately Invoked Function Expression (IIFE). This wrapping helps encapsulate the module's scope, keeping variables and functions private to the module.

4.Code Evaluation and Module Exports
After wrapping, Node.js evaluates the module’s code. During this evaluation, module.exports is set to export the module's functionality or data. This step essentially makes the module's exports available to other files.

---

5. Caching(very imp)
Importance: Caching is crucial for performance. Node.js caches the result of the require() call so that the module is only loaded and executed once.

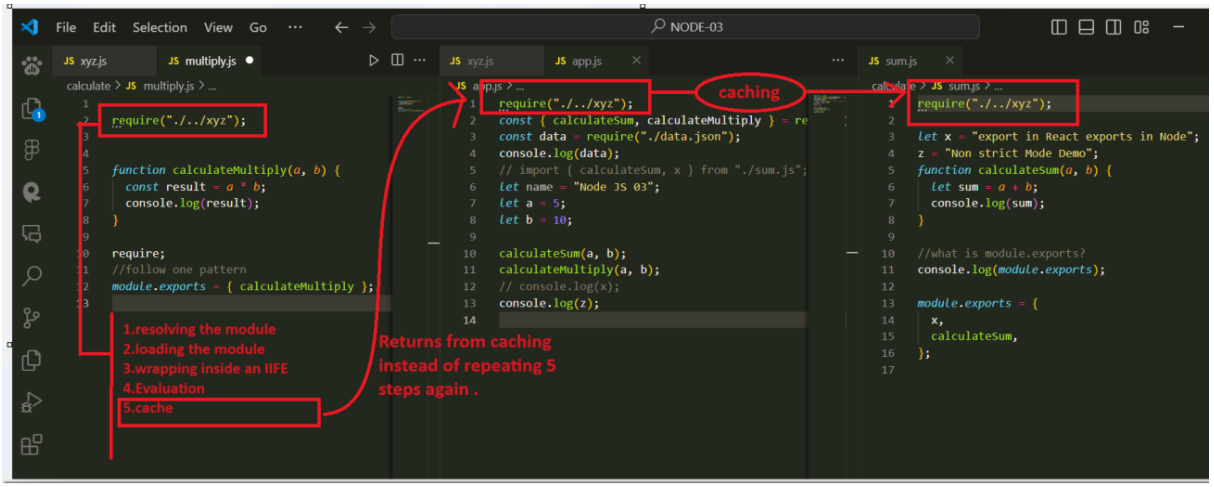
:

- **Scenario:** Suppose you have three files: sum.js , app.js , and multiply.js . All three files require a common module named xyz .
- **Initial Require:** When sum.js first requires xyz with require('./xyz') , Node.js performs the full require() process for xyz :

1. Resolving the path to xyz .
2. Loading the content of xyz
3. Wrapping the code in an IIFE.
4. Evaluating the code and setting module.exports .
5. Caching is the result.
   Node.js creates a cached entry for xyz that includes the evaluated module exports.

Subsequent Requires:

- When app.js and multiply.js later require xyz using require('./xyz') , Node.js skips the initial loading and evaluation steps. Instead, it retrieves the module from the cache.
- This means that for app.js and multiply.js , Node.js just returns the cached module.exports without going through the resolution, loading, and wrapping steps again.



Impact on Performance:

- If caching did not exist, each require('./xyz') call would repeat the full module
  loading and evaluation process. This would result in a performance overhead,
  especially if xyz is a large or complex module and is required by many files.
- With caching, Node.js efficiently reuses the module's loaded and evaluated code, significantly speeding up module resolution and reducing overhead.

GOES TO GITHUB REPO[T.S.→35]