

Episode 04: Mastering Module Exports & Requirements – Unlocking NodeJs Power!

Episode 4: Module Export & Requirements.

We all know that keeping all the Node.js code in a single file is not good practice, right? We need multiple files to create a large project , so in this chapter, we will explore how we can create those file and import concepts like modules and export requirements. lets start

A screenshot of the Visual Studio Code (VS Code) interface. The title bar says "Welcome". There are two tabs open: "JS app.js" and "JS xyz.js". The "app.js" tab is active and shows the following code:

```
1 Let name = "Node JS 03";
2 Let a = 5;
3 Let b = 10;
```

The sidebar on the left shows icons for "FOL...", "File", "Edit", "Search", and "Settings".

These two files, **app.js** and **xyz.js**, have different code that is not related to each other, so **in NodeJS we call them separate modules**.

Q: How do you make two modules work together?

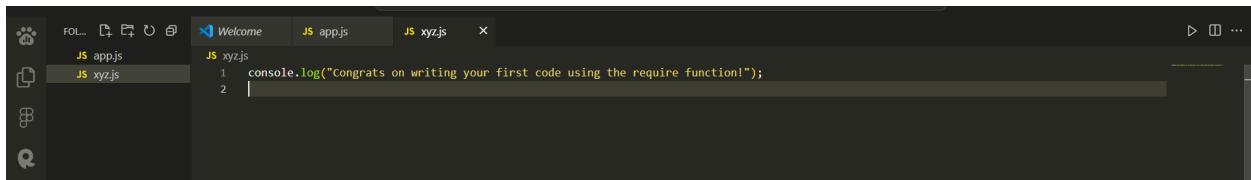
-using a **require** function

Q: What is the required function?

In Node.js, the `require()` function is a built-in function that allows you to include or require other modules into your main modules.

Now, let's write our code using the `require` function.

Task: Our objective is to execute the code written in the `xyz.js` module by running the `app.js` module.

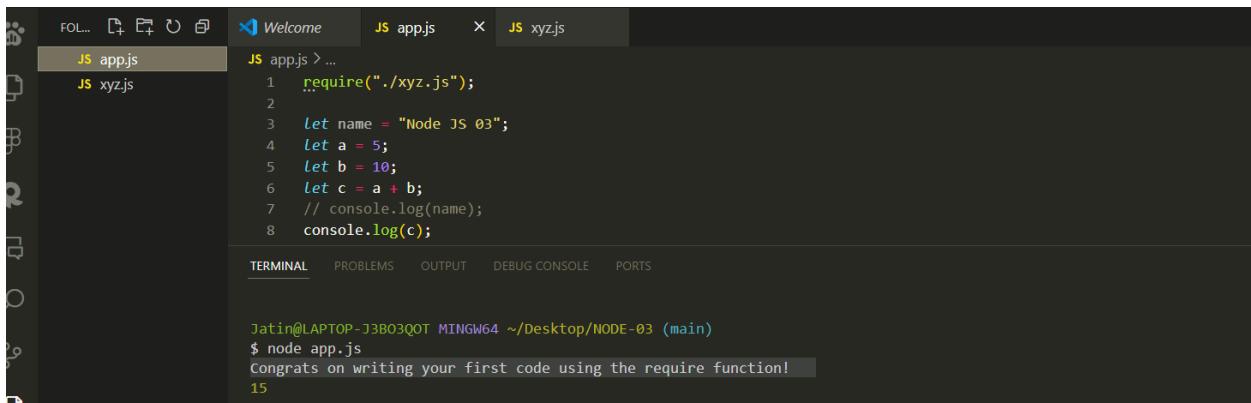


```
JS app.js
JS xyz.js
```

```
1 console.log("Congrats on writing your first code using the require function!");
2 
```

Steps:

1. Open the `app.js` module.
2. First, include the `xyz` module using the `require` function.
3. Then, run the code using Node.js. (As discussed in the last lecture, I hope you have revised it well 😊.)



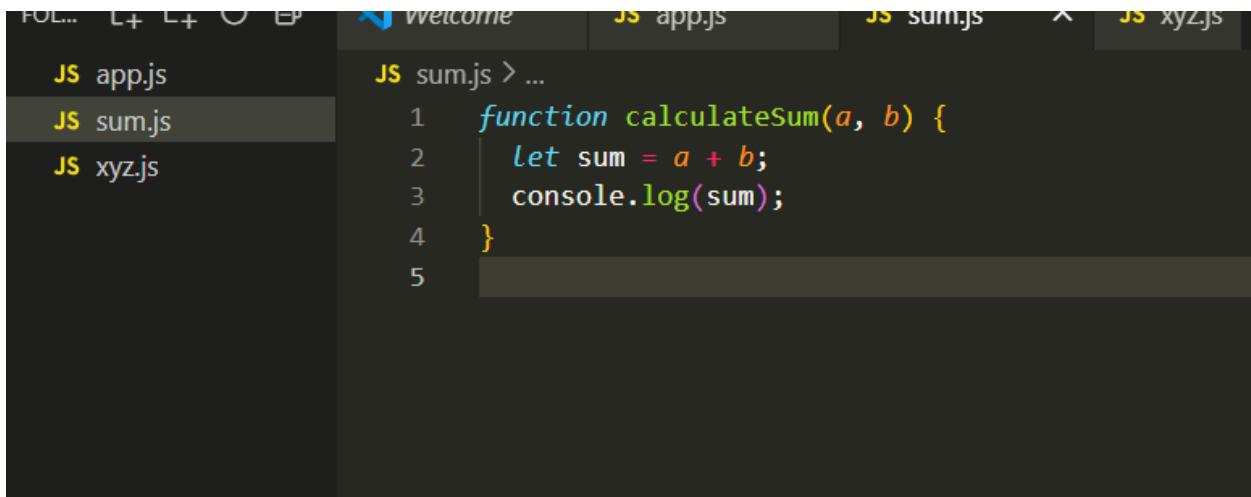
```
JS app.js > ...
1 require("./xyz.js");
2
3 let name = "Node JS 03";
4 let a = 5;
5 let b = 10;
6 let c = a + b;
7 // console.log(name);
8 console.log(c);
```

```
Jatin@LAPTOP-J3B03QOT MINGW64 ~/Desktop/NODE-03 (main)
$ node app.js
Congrats on writing your first code using the require function!
15
```

Let's look at one more example.

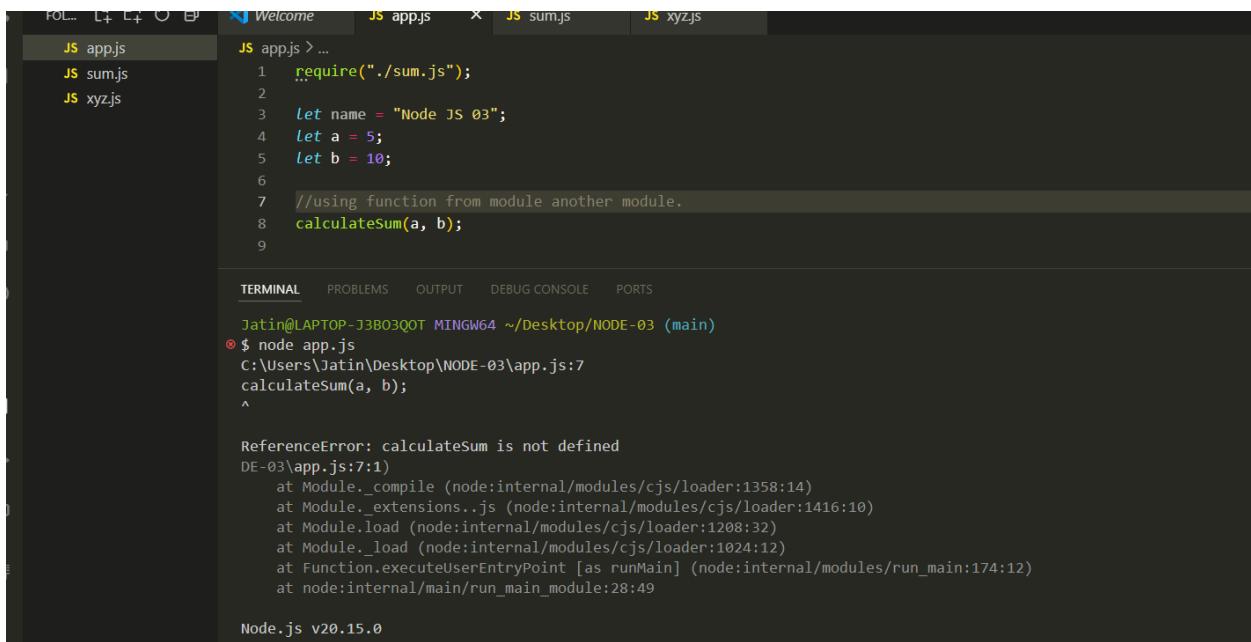
Disclosure: Before we proceed, I want to highlight that many Node.js developers may not be aware of the next concept I'm about to share, so please pay close attention.

Q: If I write a function in another module, can I use that function in a different module? Will it work or not?



```
VS Code Editor showing three files: app.js, sum.js, and xyz.js. The sum.js file is open and contains the following code:
```

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



```
VS Code Terminal output showing the execution of app.js. The terminal shows the code being run and the resulting ReferenceError.
```

```
VS Code Terminal Output:
```

```
Jatin@LAPTOP-J3B03QOT MINGW64 ~/Desktop/NODE-03 (main)
$ node app.js
C:\Users\Jatin\Desktop\NODE-03\app.js:7
  calculateSum(a, b);
^

ReferenceError: calculateSum is not defined
DE-03\app.js:7:1
    at Module._compile (node:internal/modules/cjs/loader:1358:14)
    at Module._extensions..js (node:internal/modules/cjs/loader:1416:10)
    at Module.load (node:internal/modules/cjs/loader:1208:32)
    at Module._load (node:internal/modules/cjs/loader:1024:12)
    at Function.executeUserEntryPoint [as runMain] (node:internal/modules/run_main:174:12)
    at node:internal/main/run_main_module:28:49

Node.js v20.15.0
```

Ans:

It will not work 😱

Reason(very Imp):

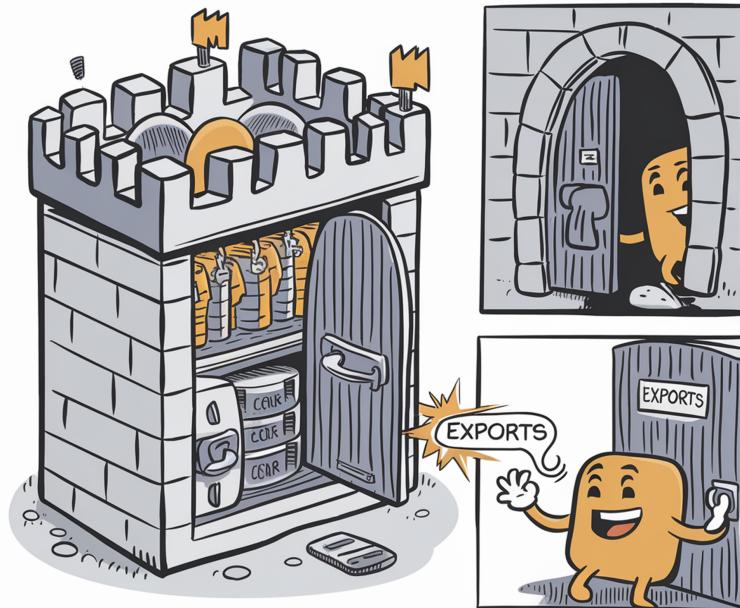
Modules protect their variables and functions from leaking by default.

Q:

So, how do we achieve that?

A:

We need to export the function using



```
module.exports
```

The screenshot shows a code editor interface with the following details:

- File Explorer:** On the left, it lists three files: `app.js`, `sum.js` (which is selected), and `xyz.js`.
- Toolbar:** At the top, there are icons for file operations like new file, open, save, and close.
- Header:** The header bar shows "Welcome" and tabs for "app.js" and "sum.js".
- Code Editor:** The main area displays the contents of `sum.js`. The code is as follows:

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

//exporting
module.exports = calculateSum;
```

But it still won't work. 🤔 Why? s

Reason: You also need to import.

The screenshot shows the VS Code interface. In the top navigation bar, there are tabs for 'Welcome', 'app.js', 'sum.js', and 'xyz.js'. The 'app.js' tab is active. The code in 'app.js' is as follows:

```
JS app.js >
1 //import syntax
2 const calculateSum = require("./sum.js");
3
4 let name = "Node JS 03";
5 let a = 5;
6 let b = 10;
7
8 //using function from another module.
9 calculateSum(a, b);
10
```

A red oval highlights the line 'const calculateSum = require("./sum.js");'. Below the code editor, the 'TERMINAL' tab is selected, showing the command prompt output:

```
Jatin@LAPTOP-33B03QQT MINGW64 ~/Desktop/NODE-03 (main)
$ node app.js
15
```

A red oval highlights the command '\$ node app.js' and the resulting output '15'.

Q: Suppose you need to export a variable, `let x = "export in React exports in Node,"` and a function, `calculateSum`. How would you do this?

A: You can export both the variable and the function by wrapping them inside an object

The screenshot shows the VS Code interface. In the top navigation bar, there are tabs for 'Welcome', 'app.js', 'sum.js', and 'xyz.js'. The 'sum.js' tab is active. The code in 'sum.js' is as follows:

```
JS sum.js > ...
1 let x = "export in React exports in Node";
2
3 function calculateSum(a, b) {
4     let sum = a + b;
5     console.log(sum);
6 }
7
8 //exporting
9 module.exports = {
10     x: x,
11     calculateSum: calculateSum,
12 };
13
```

The screenshot shows the Visual Studio Code interface. At the top, there are tabs for 'Welcome', 'app.js', 'sum.js', and 'xyz.js'. The 'app.js' tab is active, displaying the following code:

```
//import syntax 😊
const obj = require("./sum.js");
let name = "Node JS 03";
let a = 5;
let b = 10;
obj.calculatesum(a, b);
console.log(obj.x);
```

Below the editor, a navigation bar includes 'TERMINAL', 'PROBLEMS', 'OUTPUT', 'DEBUG CONSOLE', and 'PORTS'. The terminal window at the bottom shows the command line output:

```
Jatin@LAPTOP-J3B03QOT MINGW64 ~/Desktop/NODE-03 (main)
$ node app.js
15
export in React exports in Node
```

Many developers use destructuring as a common pattern to write cleaner and more efficient code. You'll encounter this technique frequently throughout your development journey 😊.



If you want to learn more about destructuring in JavaScript, refer to this link:

<https://courses.bigbinaryacademy.com/learn-javascript/object-destructuring/object-destructuring/>

```
// import syntax 😊
const { calculateSum } = require("./sum.js");
let name = "Node JS 03";
let a = 5;
let b = 10;
calculateSum(a, b);
console.log(x);
```

```
Jatin@LAPTOP-J3B03QOT MINGW64 ~/Desktop/NODE-03 (main)
$ node app.js
15
export in React exports in Node
```

Important Note:

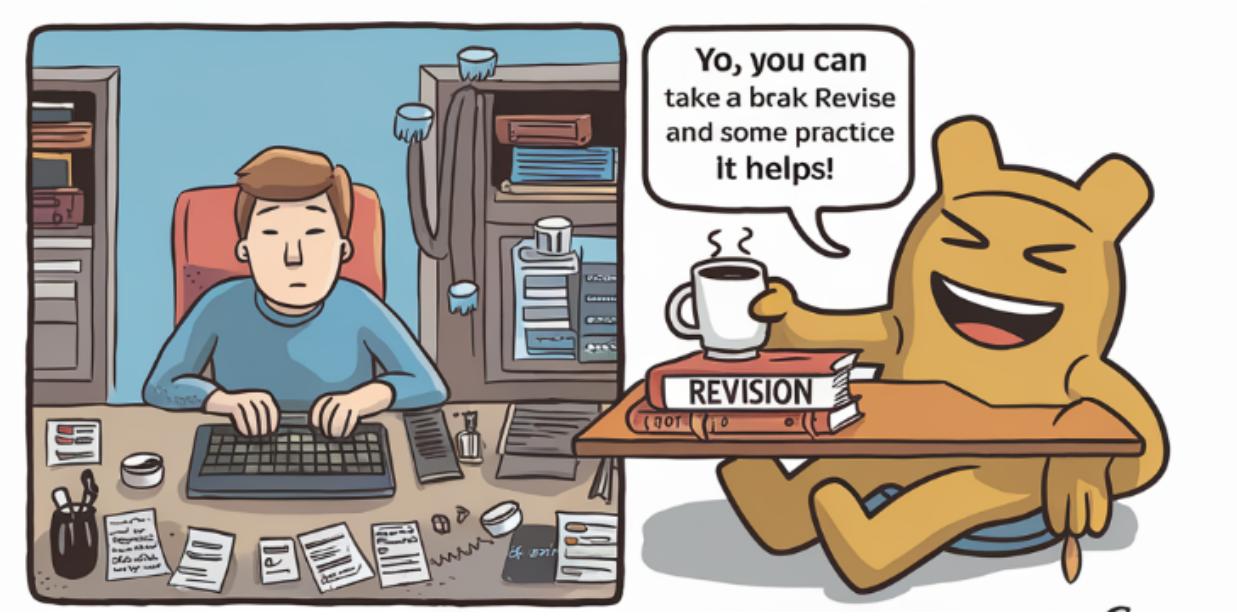
When using the following import statement:

```
const { x, calculateSum } = require("/sum");
```

you can omit the `.js` extension, and it will still work correctly. Node.js automatically resolves the file extension for you.

summary: To use private variables and functions in other modules, you need to export them. This allows other parts of your application to access and utilize those variables and functions.

Now go and practice and revise.



Welcome back! Now, let's dive into another module pattern.

We've already studied **CommonJS modules (CJS)**. Now, I'll introduce you to another type of module known as **ES modules (ESM)**, which typically use the `.mjs` extension.

Common JS / ES Modules



commonJs Modules(cjs)	Es modules (Esm) (mjs)
<ul style="list-style-type: none"> • module.exports require() • by Default used NodeJs • Older Way • synchronous • non strict <p style="color: red; margin-left: 10px;">int</p>	<ul style="list-style-type: none"> • import export • By Default used in frameworks like react , angular • Newer way • async • strict

There are two major differences between these two module systems that are important to note:

- **Synchronous vs. Asynchronous:** CommonJS requires modules in a synchronous manner, meaning the next line of code will execute only after the module has been loaded. In contrast, ES modules load modules asynchronously, allowing for more efficient and flexible code execution. This distinction is a powerful feature and an important point to remember for interviews.
- **Strict Mode:** Another significant difference is that CommonJS code runs in non-strict mode, while ES modules execute in strict mode. This means that ES modules enforce stricter parsing and error handling, making them generally safer and more reliable.

Overall, ES modules are considered better due to these advantages



- First, you need to create a new file called `package.json` (I will explain more about `package.json` later). For now, think of it as a configuration file.

To use ES modules, you must include the following in your `package.json`:

The screenshot shows the VS Code interface with the package.json file open. The sidebar on the left lists files: app.js, package.json (selected), sum.js, and xyz.js. The package.json content is:

```
1  {
2  |   "type": "module"
3  }
4
```

This setting indicates that your code will use ES module syntax.

- Syntax of Es modules

The screenshot shows the VS Code interface with the sum.js file open. The sidebar on the left lists files: app.js, sum.js (selected), and xyz.js. The sum.js content is:

```
1 let x = "export in React exports in Node";
2
3 export function calculateSum(a, b) {
4     let sum = a + b;
5     console.log(sum);
6 }
```

The screenshot shows the VS Code interface with the app.js file open. The sidebar on the left lists files: app.js (selected), sum.js, and xyz.js. The app.js content is:

```
1 //import syntax 😊
2 // const { x, calculateSum } = require("./sum.js")
3 import { calculateSum } from "./sum.js";
4 let name = "Node JS 03";
5 let a = 5;
6 let b = 10;
7
8 calculateSum(a, b);
9 console.log(x);
10
```

- export import multiple

A screenshot of the Visual Studio Code interface. The left sidebar shows files: app.js (selected), sum.js (highlighted), and xyz.js. The main editor area shows the contents of sum.js:

```
1  export let x = "export in React exports in Node";
2
3  export function calculateSum(a, b) {
4      let sum = a + b;
5      console.log(sum);
6 }
```

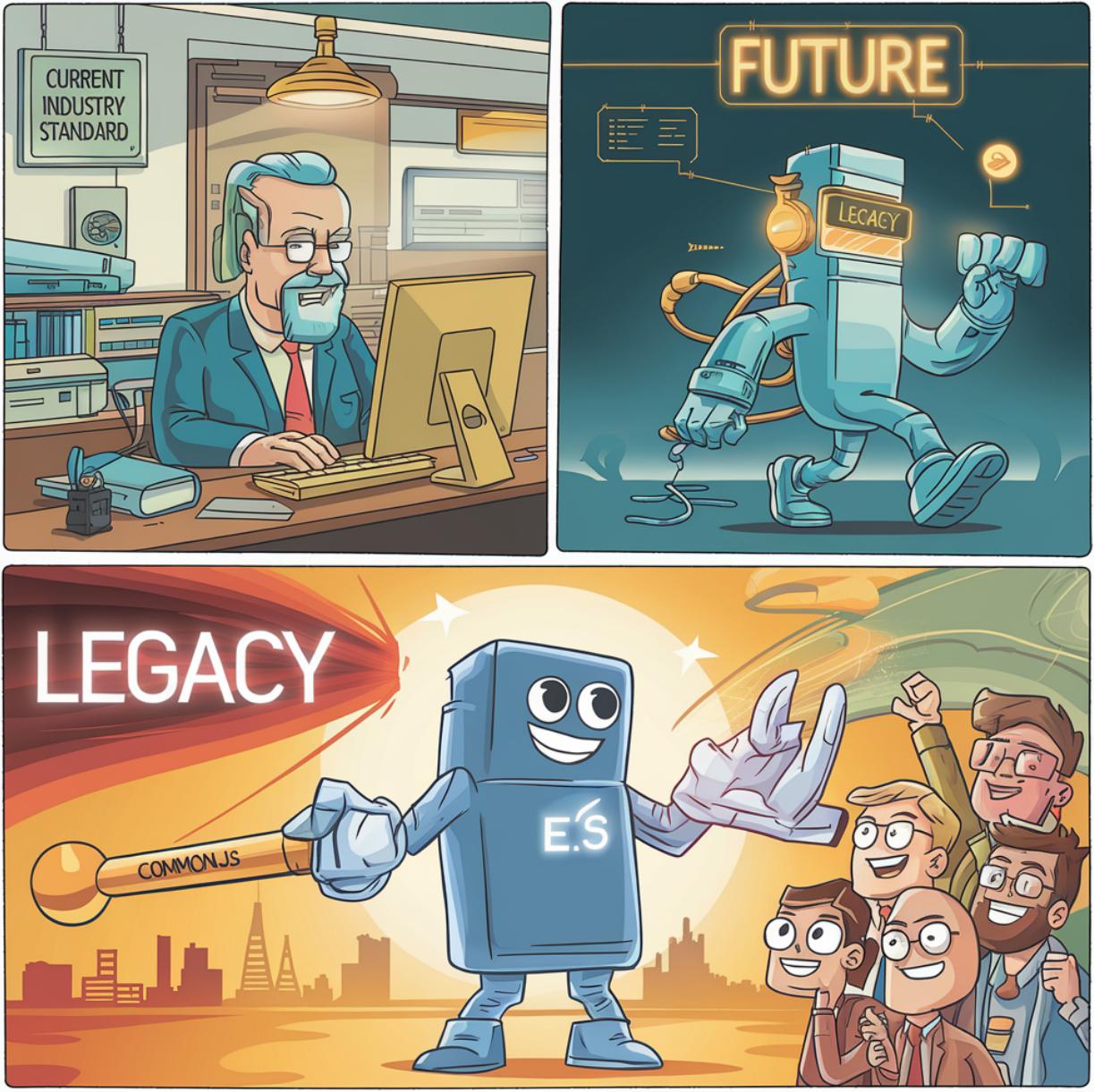
A screenshot of the Visual Studio Code interface. The left sidebar shows files: app.js (selected), package.json, sum.js, and xyz.js. The main editor area shows the contents of app.js:

```
1 //import syntax 😞
2 // const { x, calculateSum } = require("./sum.js");
3 import { calculateSum, x } from "./sum.js";
4 let name = "Node JS 03";
5 let a = 5;
6 let b = 10;
7
8 calculateSum(a, b);
9 console.log(x);
10
```

The terminal below shows the output of running the script:

```
Jatin@LAPTOP-J3B03QQT MINGW64 ~/Desktop/NODE-03 (main)
$ node app.js
15
export in React exports in Node
```

In the industry, you will still find CommonJS modules being used; however, in the next 2 to 3 years, there is expected to be a significant shift towards ES modules.



Let me show you a demo of strict mode and non-strict mode in case your interviewer asks about it—this will surely impress them!

- In a CommonJS module, you can define a variable without using `var`, `let`, or `const`, and the code will execute without throwing an error because it operates in non-strict mode.

The screenshot shows a VS Code interface with the following details:

- File Explorer:** Shows files: app.js, sum.js (selected), xyz.js.
- Editor:** sum.js content:

```
1 let x = "export in React exports in Node";
2 z = "Non strict Mode Demo";
3 function calculateSum(a, b) {
4     let sum = a + b;
5     console.log(sum);
6 }
7
8 // exporting;
9 module.exports = {
10     x,
11     calculateSum,
12 }
```
- Terminal:** Output of running node app.js:

```
Jatin@LAPTOP-J3B03QOT MINGW64 ~/Desktop/NODE-03 (main)
$ node app.js
15
export in React exports in Node
Non strict Mode Demo
```

- However, in an ES module, if you try to execute the same code, it will throw an error because ES modules run in strict mode. In strict mode, you cannot define variables without declaring them first.

The screenshot shows a VS Code interface with the following details:

- File Explorer:** Shows files: app.js, package.json, sum.js, and xyz.js.
- Code Editor:** sum.js file open, containing the following code:

```
1  export let x = "export in React exports in Node";
2  z = "strict Mode Demo";
3  export function calculateSum(a, b) {
4    let sum = a + b;
5    console.log(sum);
6  }
7
8 //exporting
9 // module.exports = {
10 //   x,
11 //   calculateSum,
12 // };
```
- Terminal:** Node.js v20.15.0 running, output:

```
Jatin@LAPTOP-J3B03QOT MINGW64 ~/Desktop/NODE-03 (main)
$ node app.js
file:///C:/Users/Jatin/Desktop/NODE-03/sum.js:2
z = "strict Mode Demo";
^
ReferenceError: z is not defined
  at file:///C:/Users/Jatin/Desktop/NODE-03/sum.js:2:3
  at ModuleJob.run (node:internal/modules/esm/module_job:222:25)
  at async ModuleLoader.import (node:internal/modules/esm/loader:316:24)
  at async asyncRunEntryPointWithESMLoader
  (node:internal/modules/run_main:123:5)

Node.js v20.15.0
```

Q: What is `module.exports` ?

- `module.exports` is an empty object by default.

A screenshot of the Visual Studio Code interface. The left sidebar shows files: app.js, sum.js, and xyz.js. The main editor area shows the contents of sum.js:

```
JS sum.js > [?] <unknown>
1 let x = "export in React exports in Node";
2 z = "Non strict Mode Demo";
3 function calculateSum(a, b) {
4     let sum = a + b;
5     console.log(sum);
6 }
7
8 //what is module.exports?
9 console.log(module.exports);
10
11 module.exports = {
12     x,
13     calculateSum,
14 };
15
```

The terminal at the bottom shows the output of running node app.js:

```
Jatin@LAPTOP-J3B03QOT MINGW64 ~/Desktop/NODE-03 (main)
$ node app.js
{}
15
export in React exports in Node
Non strict Mode Demo
```

Another common pattern you will encounter is that, instead of writing:

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

developers may prefer to write it like this:

```
module.exports.x = x;
module.exports.calculateSum = calculateSum;
```

The screenshot shows a dark-themed VS Code interface. In the top left, there's a file tree with files: app.js, sum.js, xyz.js. The sum.js file is currently selected. In the main editor area, the code for sum.js is displayed:

```
JS sum.js    X JS app.js    JS xyz.js
JS sum.js > ...
3   function calculateSum(a, b) {
6   }
7
8 //what is module.exports?
9 console.log(module.exports);
10
11 // module.exports = {
12 //   x,
13 //   calculateSum,
14 // };
15 module.exports.x = x;
16 module.exports.calculateSum = calculateSum;
17
```

Below the editor is a tab bar with TERMINAL, PROBLEMS, OUTPUT, DEBUG CONSOLE, and PORTS. The TERMINAL tab is active, showing the following terminal output:

```
Jatin@LAPTOP-J3B03QOT MINGW64 ~/Desktop/NODE-03 (main)
$ node app.js
{}
15
export in React exports in Node
Non strict Mode Demo
```

One more common pattern → Nested Modules

CREATE folder → calculate inside it create two files, **sum.js** and **multiply.js**

The screenshot shows a dark-themed VS Code interface. In the top left, there's a file tree with a folder named 'calculate' containing files: multiply.js, sum.js. The sum.js file is currently selected. In the main editor area, the code for sum.js is displayed:

```
JS sum.js    X JS multiply.js    JS app.js    JS xyz.js
calculate > JS sum.js > ...
1 let x = "export in React exports in Node";
2 z = "Non strict Mode Demo";
3 function calculateSum(a, b) {
4   let sum = a + b;
5   console.log(sum);
6 }
7
8 //what is module.exports?
9 console.log(module.exports);
10
11 module.exports = {
12   x,
13   calculateSum,
14 };
15
```

The screenshot shows the VS Code interface with the following details:

- File Explorer sidebar: A folder named "calculate" containing files: multiply.js, sum.js, app.js, and xyz.js.
- Code Editor: The active file is multiply.js, which contains the following code:

```
function calculateMultiply(a, b) {
  const result = a * b;
  console.log(result);
}

//follow one pattern
module.exports = { calculateMultiply };
```
- Tab Bar: JS sum.js, JS multiply.js (active), JS app.js, JS xyz.js.

The screenshot shows the VS Code interface with the following details:

- File Explorer sidebar: A folder named "calculate" containing files: multiply.js, sum.js, app.js (active), and xyz.js.
- Code Editor: The active file is app.js, which contains the following code:

```
// import syntax 😊
const { x, calculateSum } = require("./calculate/sum");
const { calculateMultiply } = require("./calculate/multiply");
// import { calculateSum, x } from "./sum.js";
let name = "Node JS 03";
let a = 5;
let b = 10;

calculateSum(a, b);
calculateMultiply(a, b);
console.log(x);
console.log(z);
```
- Tab Bar: JS sum.js, JS multiply.js, JS app.js (active), JS xyz.js.
- Terminal: Shows the command \$ node app.js and its output:

```
Jatin@LAPTOP-J3B03Q0T MINGW64 ~/Desktop/NODE-03 (main)
$ node app.js
{}
15
50
export in React exports in Node
Non strict Mode Demo
```
- Bottom navigation bar: TERMINAL, PROBLEMS, OUTPUT, DEBUG CONSOLE, PORTS.

one more pattern

Let's create an `index.js` file in our `calculate` folder.

The screenshot shows a code editor with a dark theme. On the left, there's a sidebar with a 'calculate' folder containing files: index.js, multiply.js, sum.js, app.js, and xyz.js. The main area shows the content of index.js:

```
1 const { calculateMultiply } = require("./multiply");
2 const { calculateSum } = require("./sum");
3 
4 module.exports = [ calculateMultiply, calculateSum ];
5
```

In the

app.js file, you simply need to require calculateSum and calculateMultiply from the calculate folder. Here's how you can do it:

The screenshot shows a code editor with a dark theme. On the left, there's a sidebar with a 'calculate' folder containing files: index.js, multiply.js, sum.js, app.js, and xyz.js. The main area shows the content of app.js:

```
1 // import syntax 😊
2 // const { x, calculateSum } = require("./calculate/sum");
3 
4 // const { calculateMultiply } = require("./calculate/multiply");
5 // instead we just require direct from calculate folder
6 const { calculateSum, calculateMultiply } = require("./calculate");
7 // import { calculateSum, x } from "./sum.js";
8 let name = "Node JS 03";
9 let a = 5;
10 let b = 10;
11 
12 calculateSum(a, b);
13 calculateMultiply(a, b);
```

Annotations with red arrows highlight the imports from the calculate folder: 'require("./calculate/sum")' and 'require("./calculate/multiply")'. Another annotation highlights the direct import statement 'const { calculateSum, calculateMultiply } = require("./calculate");'.

At the bottom, a terminal window shows the output of running node app.js:

```
Jatin@LAPTOP-J3B03QOT MINGW64 ~/Desktop/NODE-03 (main)
$ node app.js
{}
15
50
Non strict Mode Demo
```

But why are we doing all this? When you have a large project with hundreds of files, it's beneficial to group related files together and create separate modules. In this case, the calculate folder serves as a new module that encapsulates related functionality, such as the calculateSum and calculateMultiply functions.

By organizing your code into modules, you improve maintainability, readability, and scalability, making it easier to manage and navigate your codebase.



One last thing

Suppose you have a `data.json` file, and you want to import it into your JavaScript code. You can do this easily using `require`. Here's how you can import the JSON file:

The screenshot shows a file tree on the left with the following structure:

- FOL...
- calculate (expanded):
 - index.js
 - multiply.js
 - sum.js
 - app.js
- { } data.json
- xyz.js

The right side shows the content of the data.json file:

```
{ } data.json > abc city
1 {
2   "name": "ranbir kapoor",
3   "city": "Mumbai",
4   "country": "India"
5 }
6
```

The screenshot shows a file tree on the left with the following structure:

- FOL...
- calculate (expanded):
 - index.js
 - multiply.js
 - sum.js
 - app.js
- { } data.json
- xyz.js

The right side shows the content of the app.js file:

```
JS sum.js JS index.js JS multiply.js JS app.js X { } data.json JS xyz.js
JS app.js > ...
1 // import syntax 😊
2
3 const { calculateSum, calculateMultiply } = require("./calculate");
4 const data = require("./data.json");
5 console.log(data);
6 // import { calculateSum, x } from "./sum.js";
7 let name = "Node JS 03";
8 let a = 5;
9 let b = 10;
10
11 calculateSum(a, b);
12 calculateMultiply(a, b);
13 // console.log(x);
```

The terminal at the bottom shows the output of running the app.js script:

```
Jatin@LAPTOP-J3B03Q0T MINGW64 ~/Desktop/NODE-03 (main)
$ node app.js
{}
{ name: 'ranbir kapoor', city: 'Mumbai', country: 'India' }
15
50
Non strict Mode Demo
```

- There are some modules that are built into the core of Node.js, one of which is the `util` module. You can import it like this:

```
const util = require('node:util');
```

- The `util` object contains a variety of useful functions and properties.

In general, a module can be a single file or a folder. A module is essentially a collection of JavaScript code that is private to itself and organized separately. If you want to export something from a module, you can use `module.exports` to expose the desired functionality to other parts of your application

