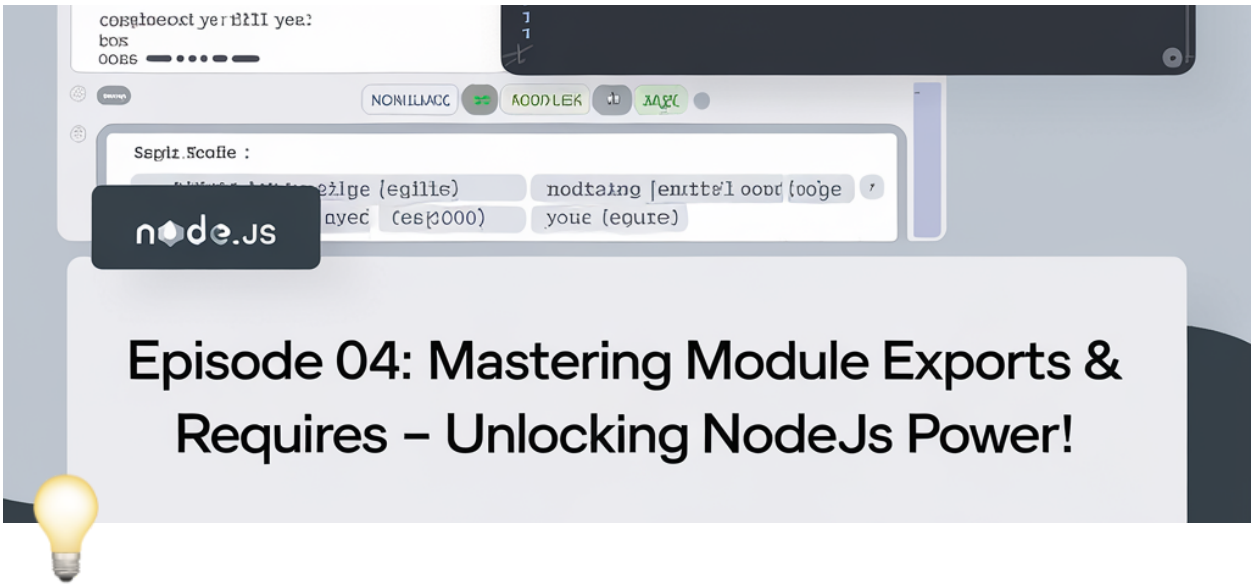
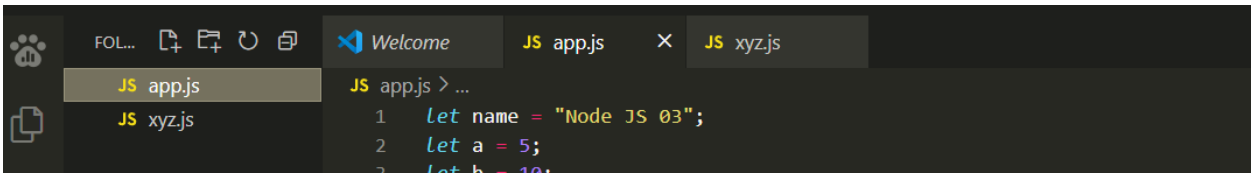


Episode 4: Module Export & Requirements.

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



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?

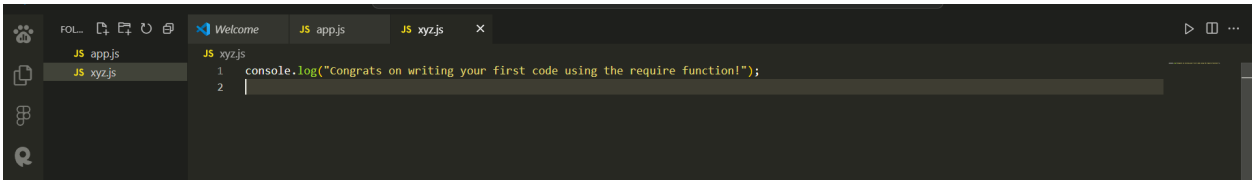
-using a **require function**

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.

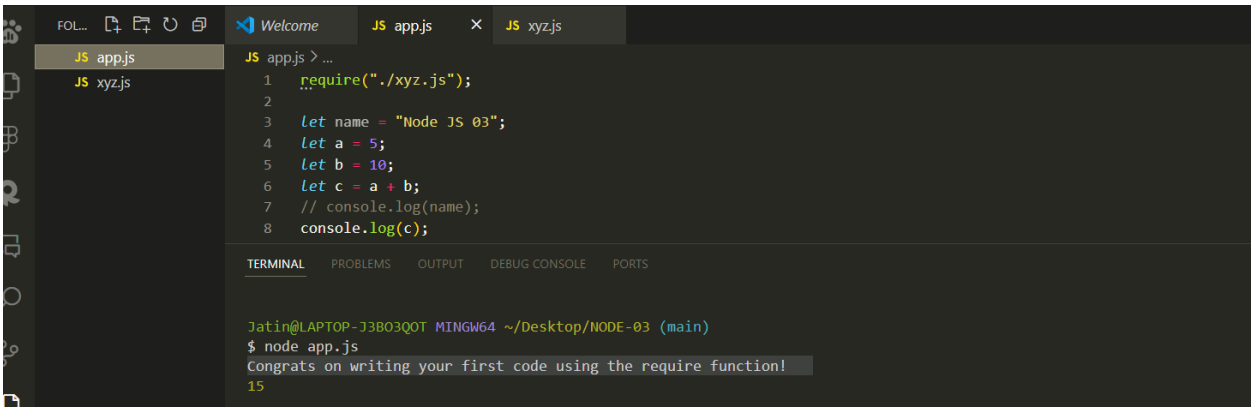
Now, let's write our code using the `require` function.

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



Steps:

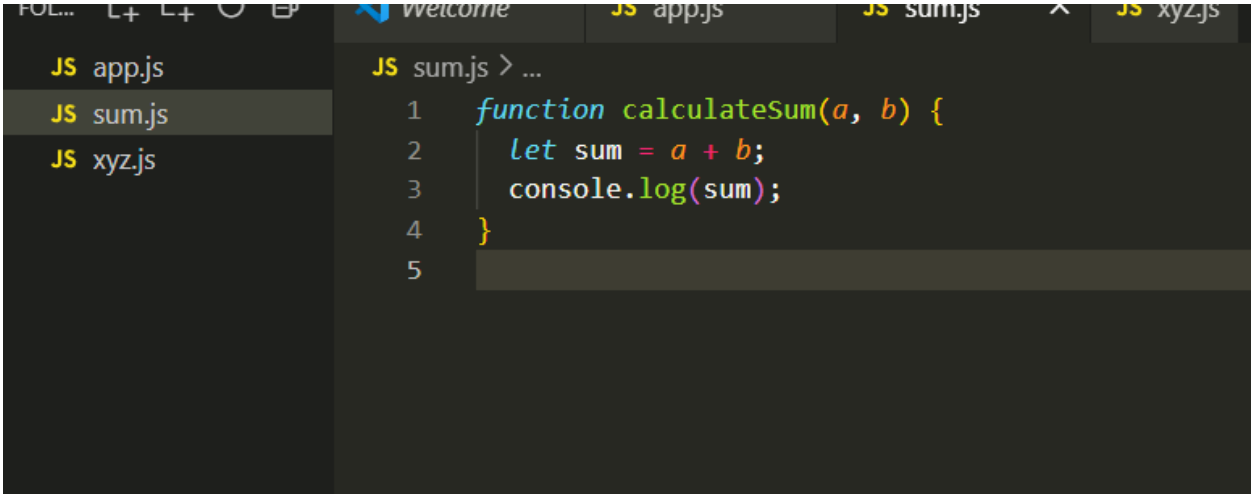
1. Open the `app.js` module.
2. First, include the `xyz` module using the `require` function.
3. Then, run the code using Node.js. (As discussed in the last lecture, I hope you have revised it well🙄.)

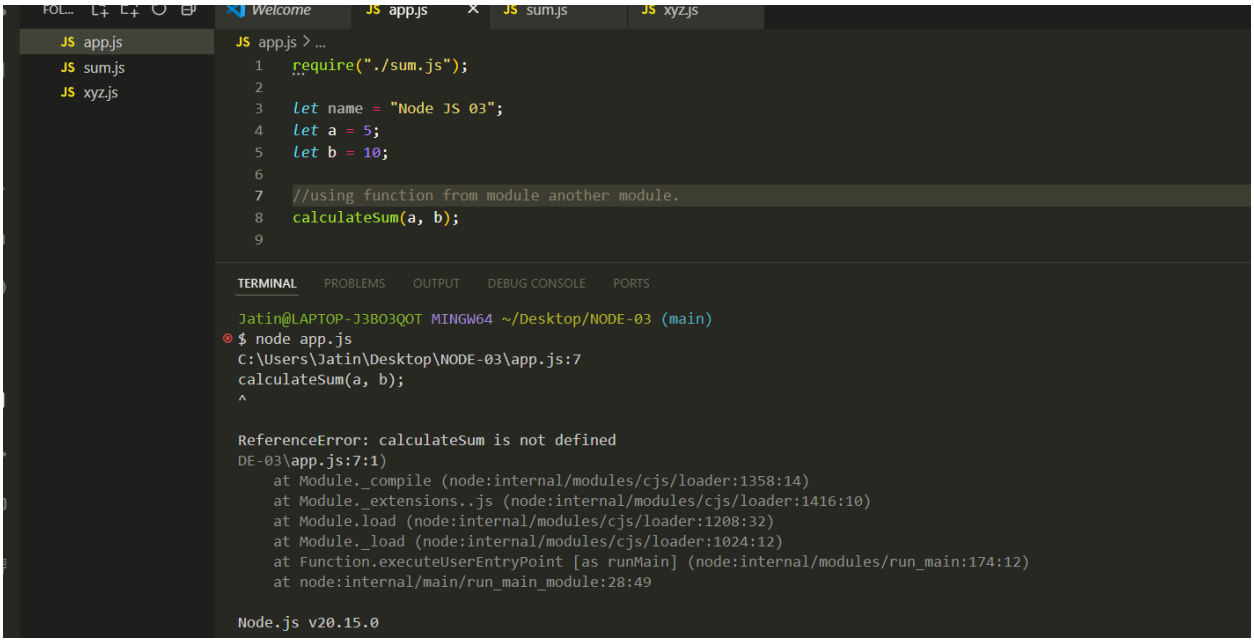


Let’s look at one more example.

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?





Ans:

It will not work 🤯

Reason(very Imp):

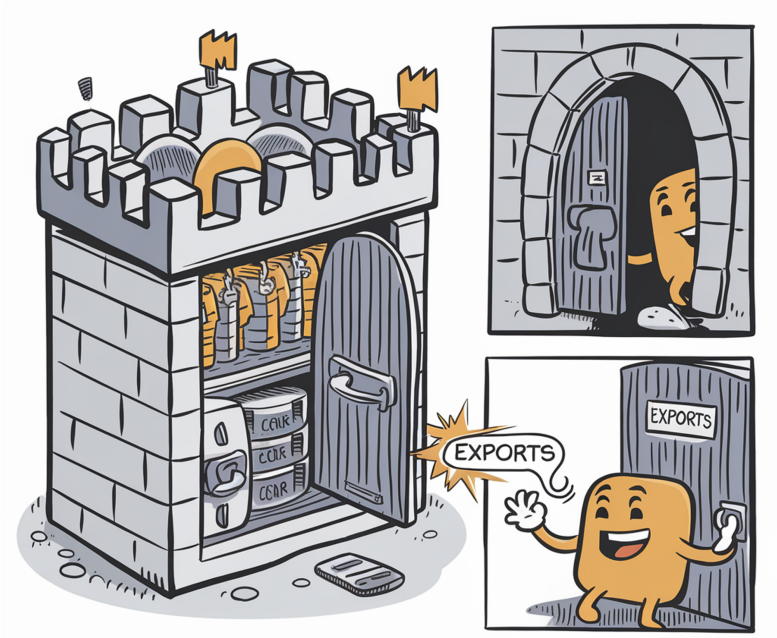
Modules protect their variables and functions from leaking by default.

Q:

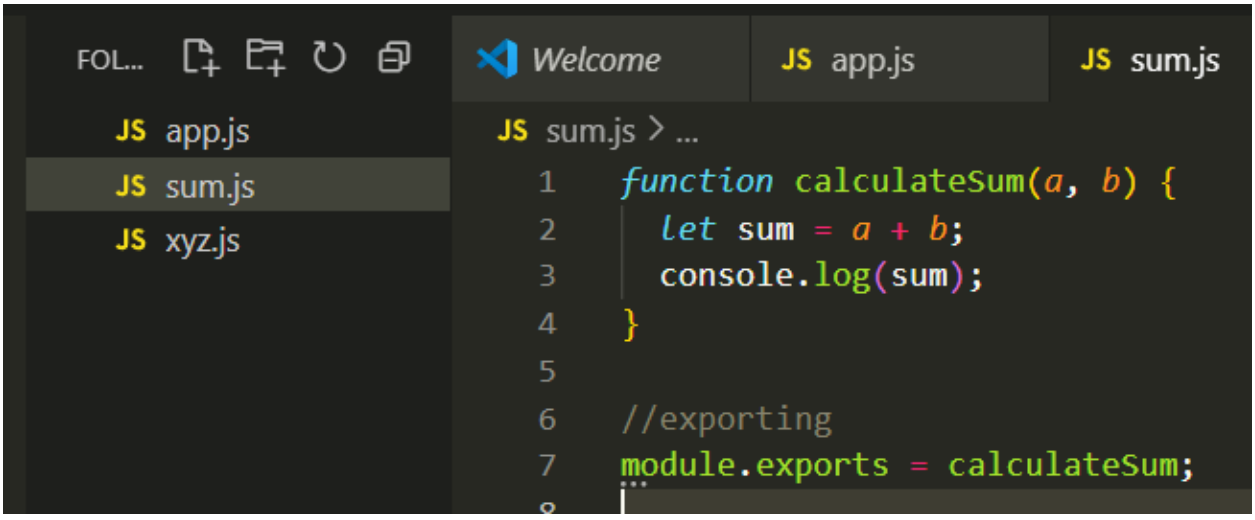
So, how do we achieve that?

A:

We need to export the function using

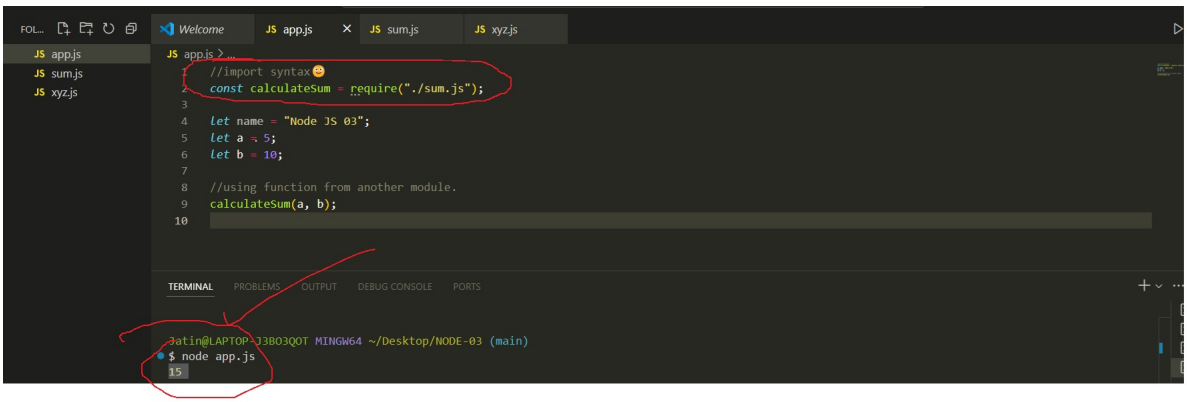



```
module.exports
```



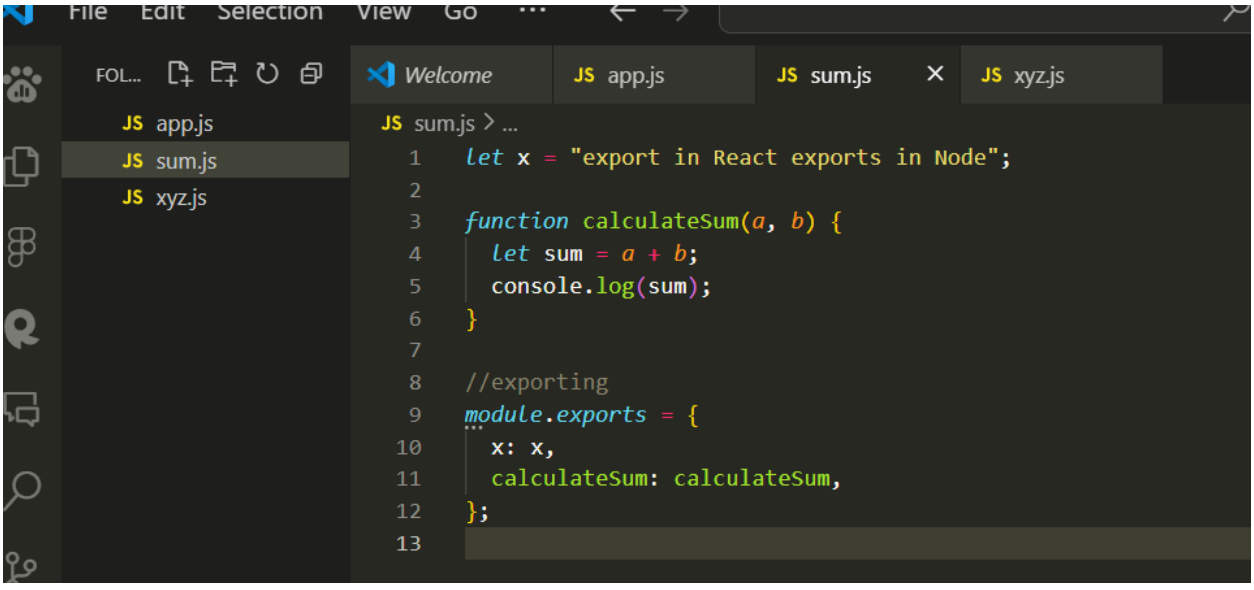
But it still won't work.🤷 Why?s

Reason: You also need to import.



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?

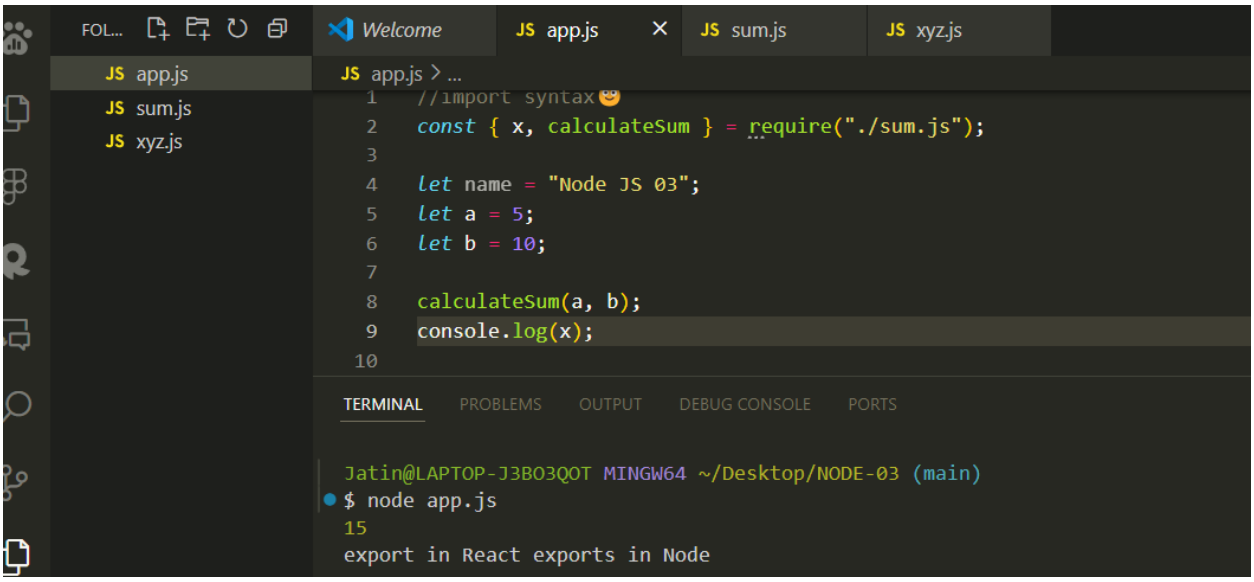
A: You can export both the variable and the function by wrapping them inside an object



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/



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.

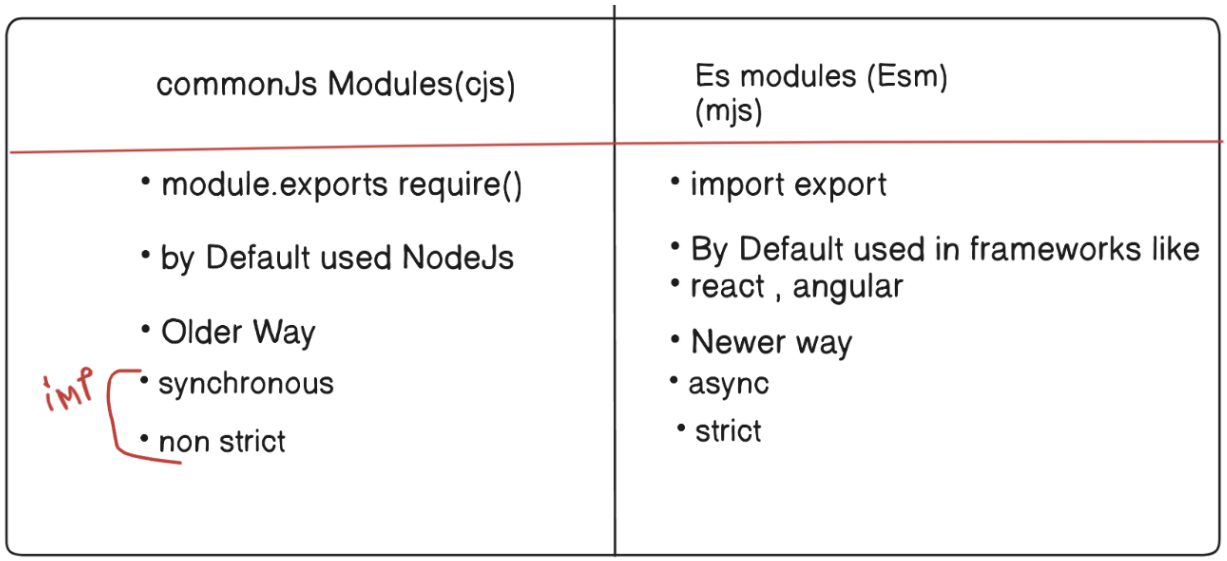
summary: *To use private variables and functions in other modules, you need to export them. This allows other parts of your application to access and utilize those variables and functions.*

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



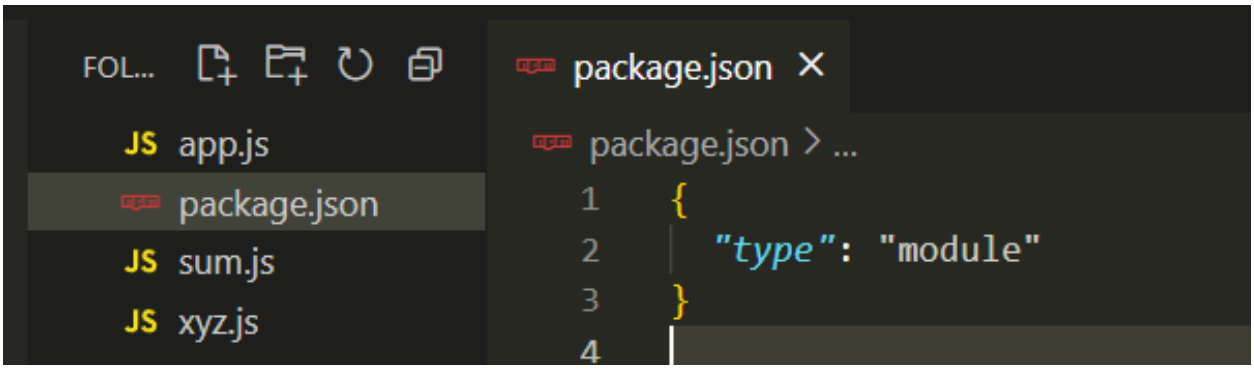
There are two major differences between these two module systems that are important to note:

- **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

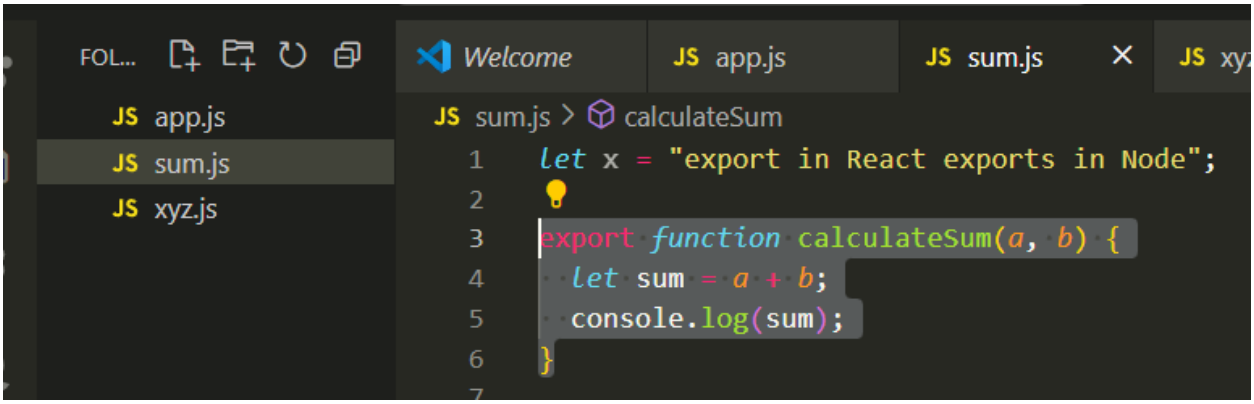
- First, you need to create a new file called `package.json` (I will explain more about `package.json` later). For now, think of it as a configuration file.

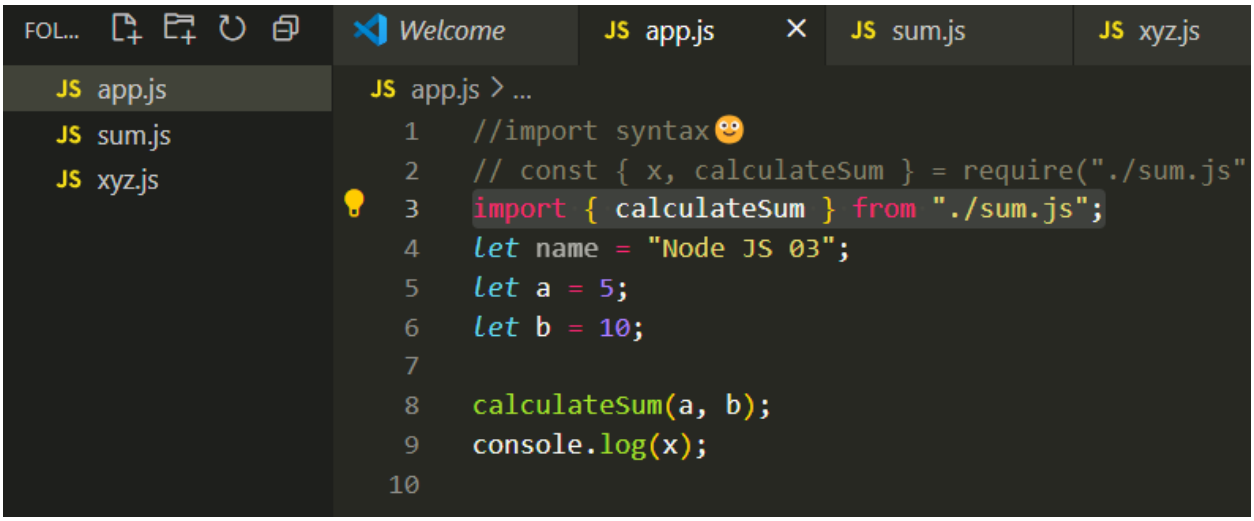
To use ES modules, you must include the following in your `package.json`:



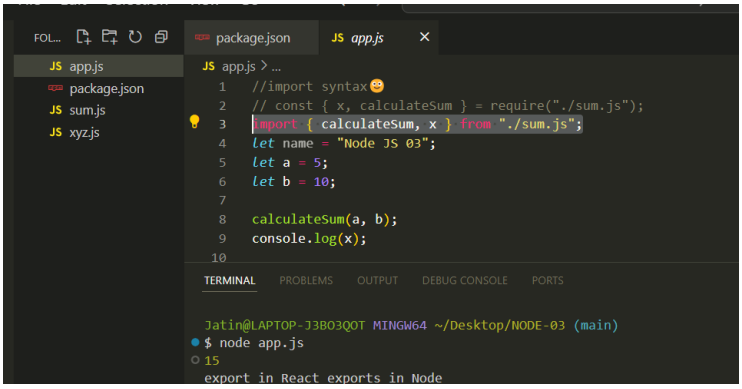
This setting indicates that your code will use ES module syntax.

- Syntax of Es modules

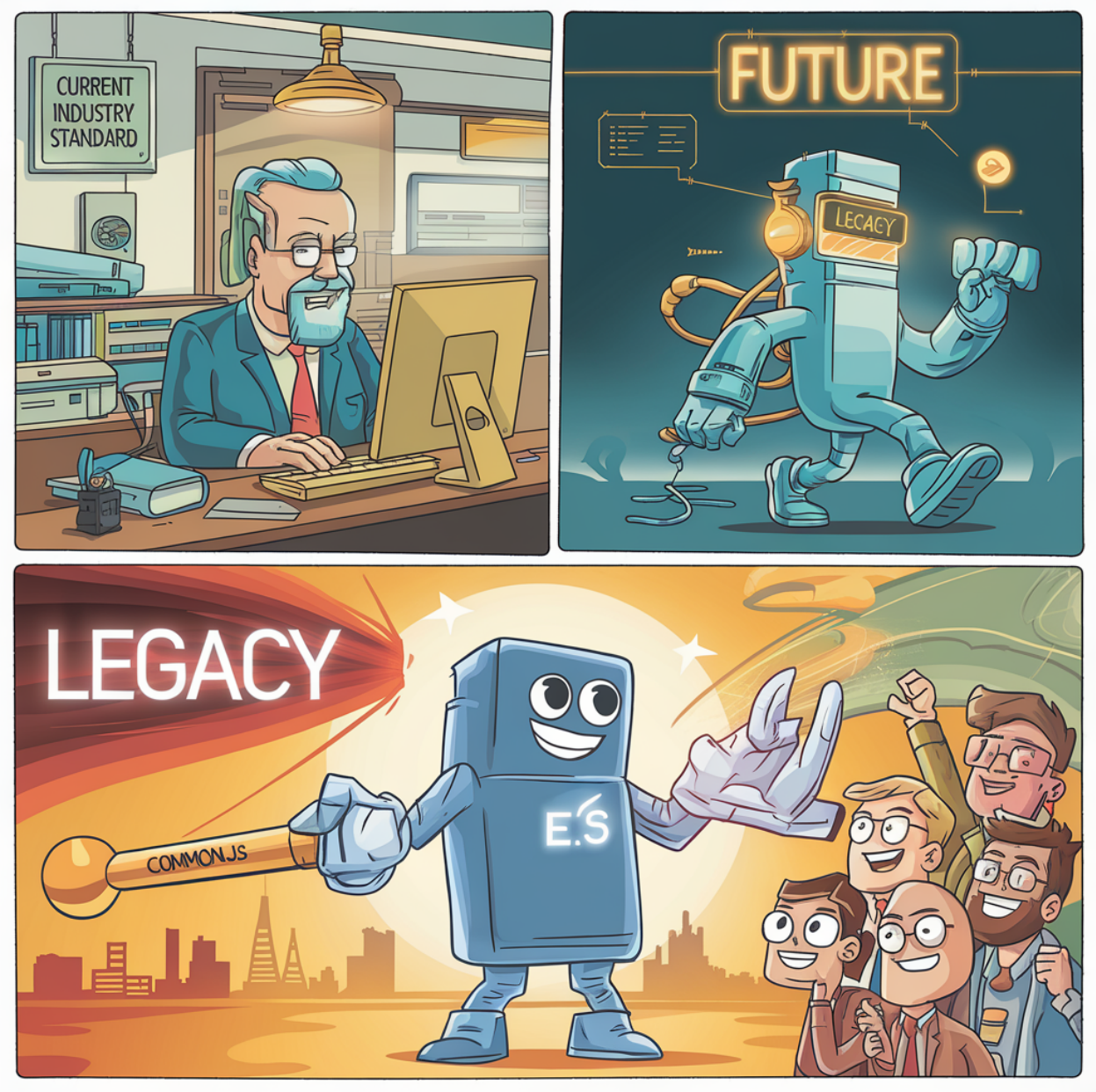




- export import multiple

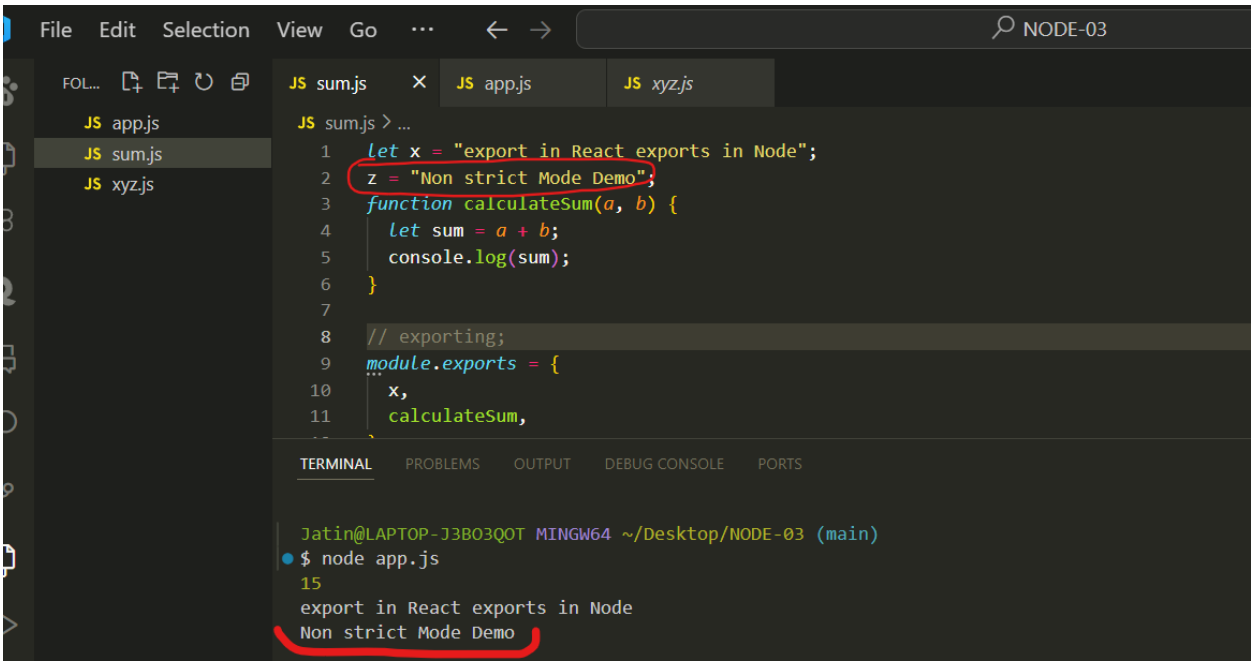


In the industry, you will still find CommonJS modules being used; however, in the next 2 to 3 years, there is expected to be a significant shift towards ES modules.

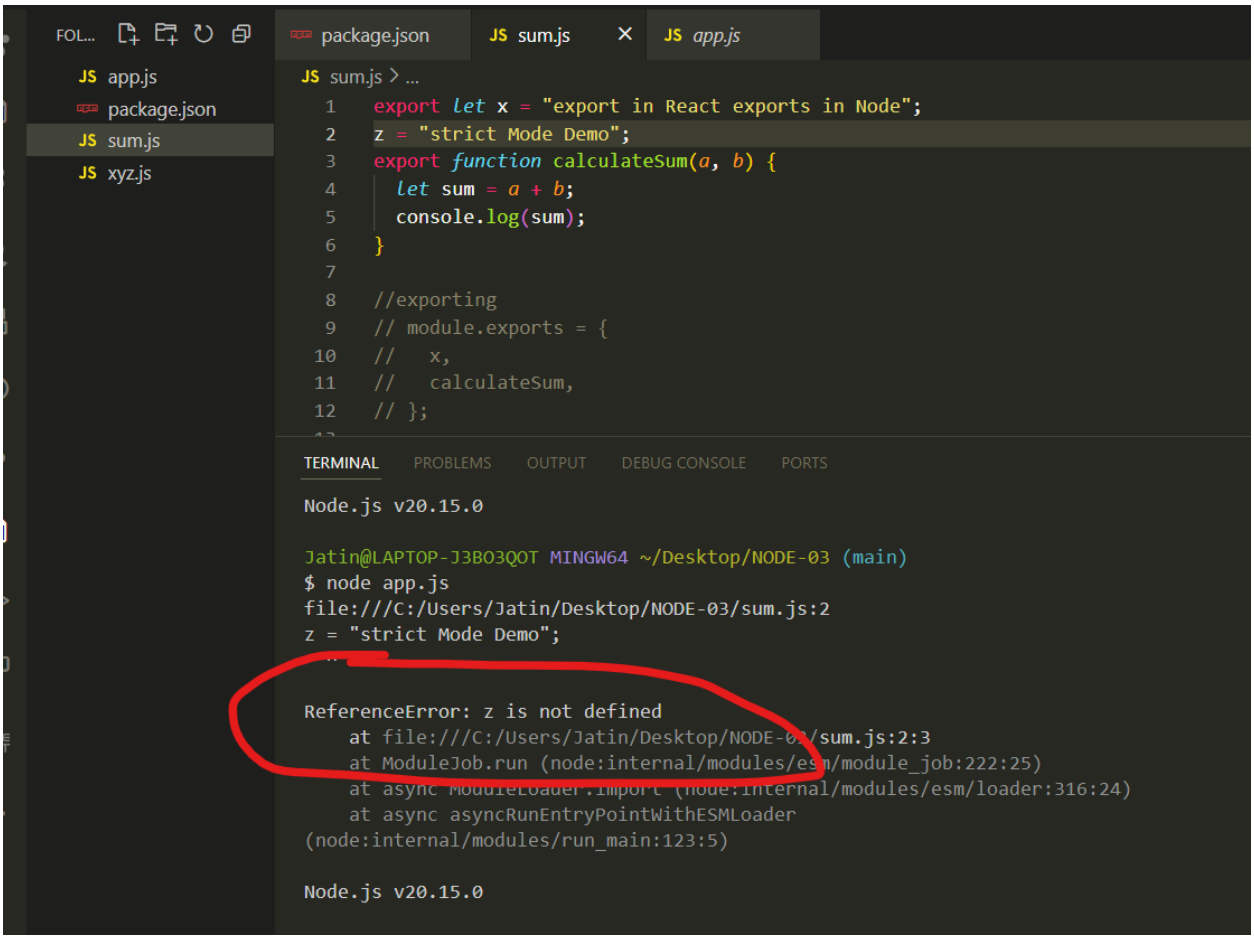


Let me show you a demo of strict mode and non-strict mode in case your interviewer asks about it—this will surely impress them!

- 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.

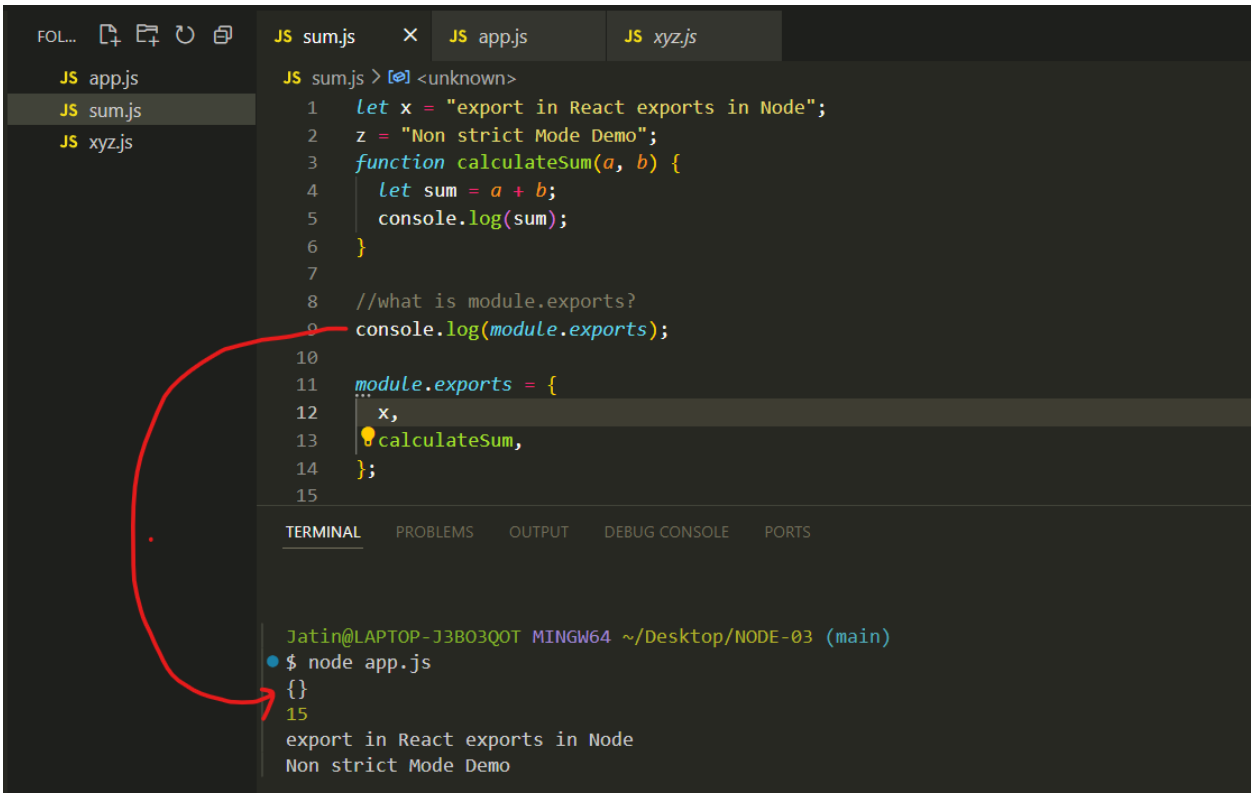


- However, in an ES module, if you try to execute the same code, it will throw an error because ES modules run in strict mode. In strict mode, you cannot define variables without declaring them first.



Q: What is `module.exports`?

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

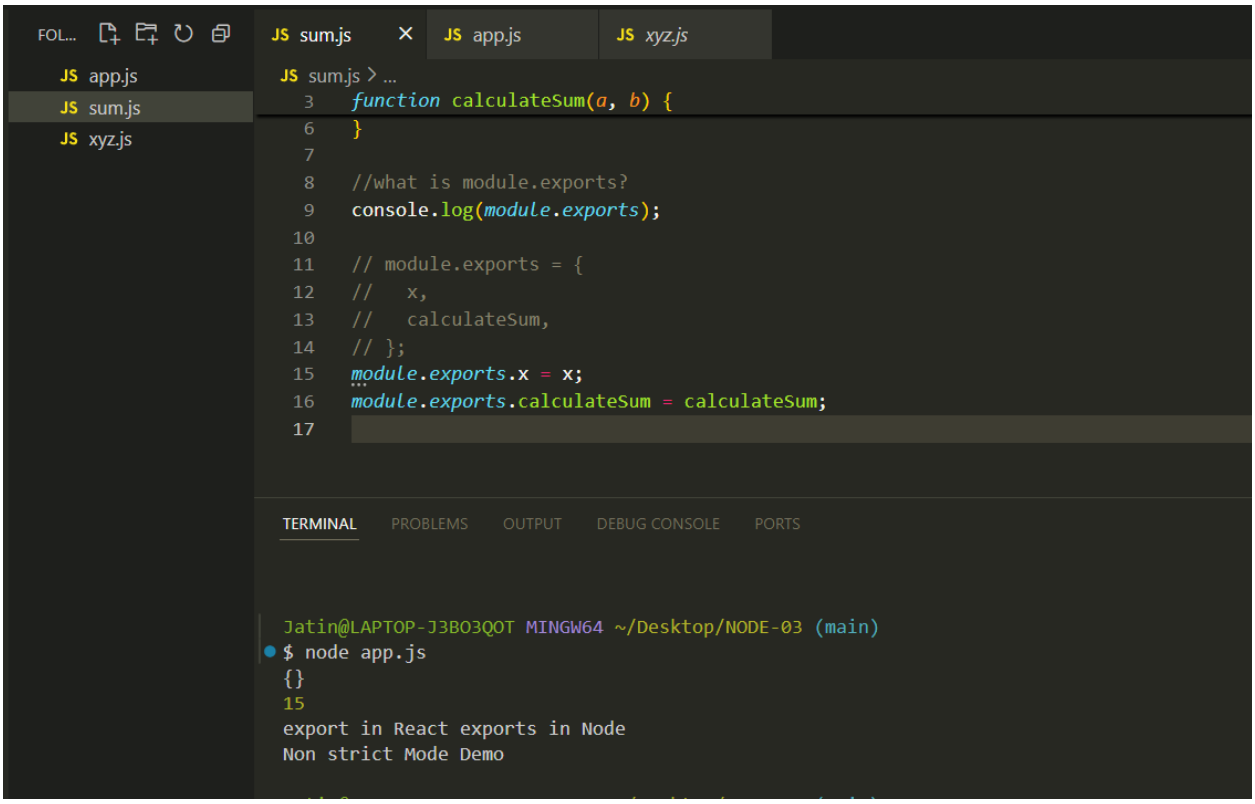


Another common pattern you will encounter is that, instead of writing:

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

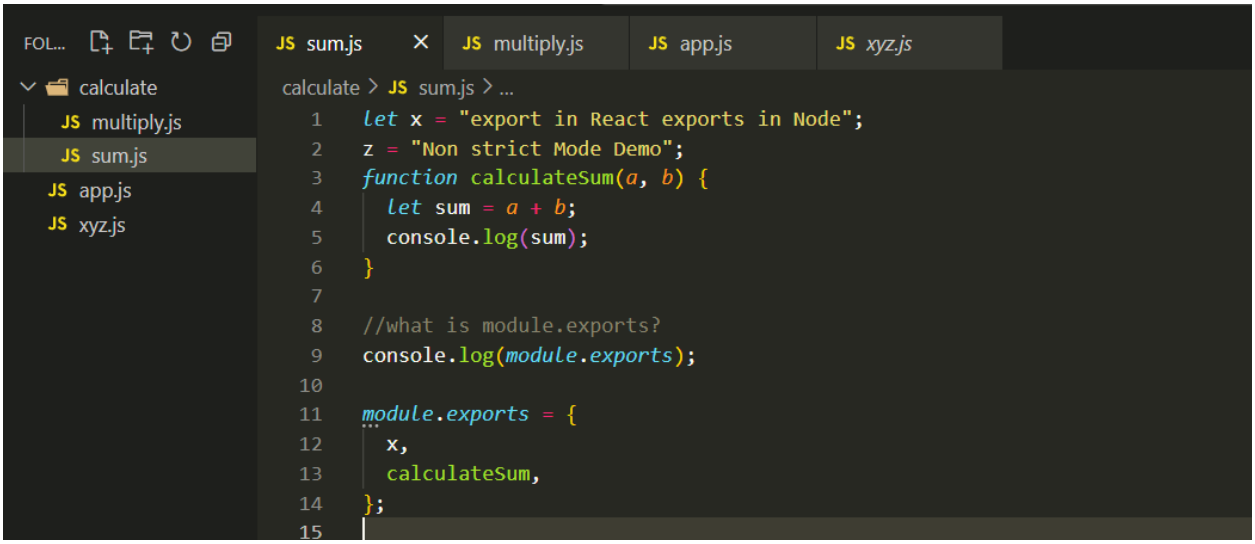
developers may prefer to write it like this:

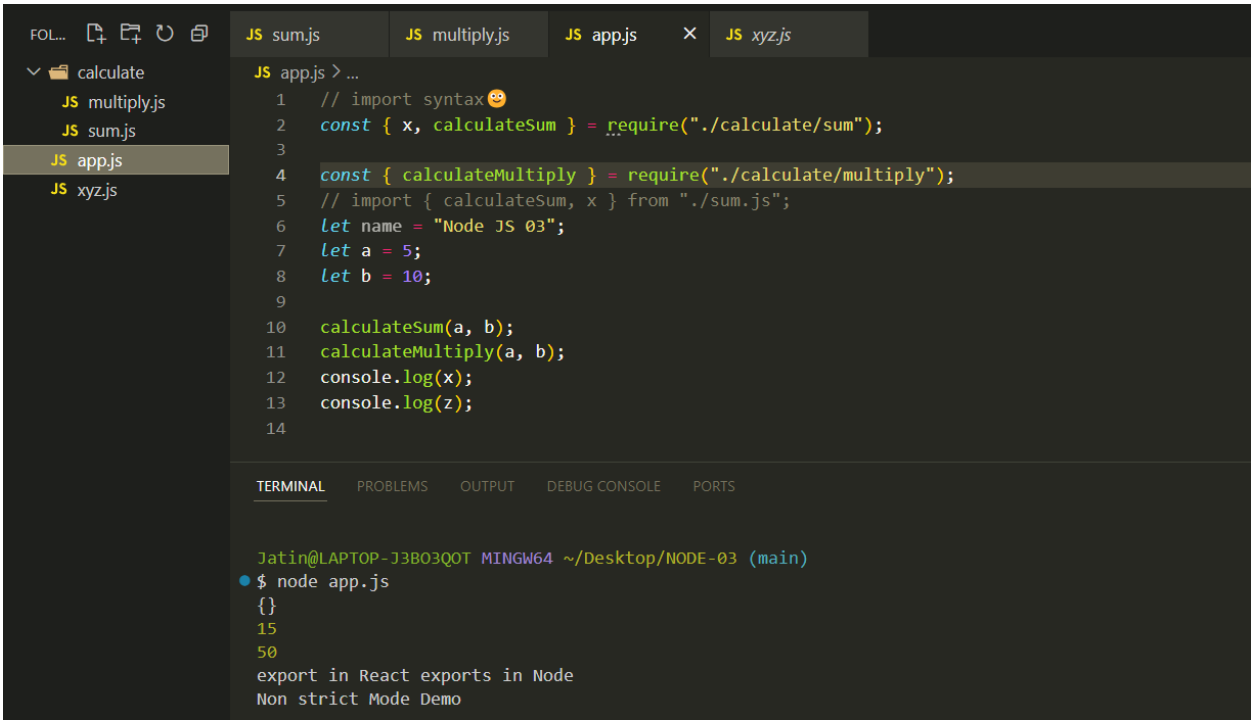
```
module.exports.x = x;
module.exports.calculateSum = calculateSum;
```



One more common pattern → Nested Modules

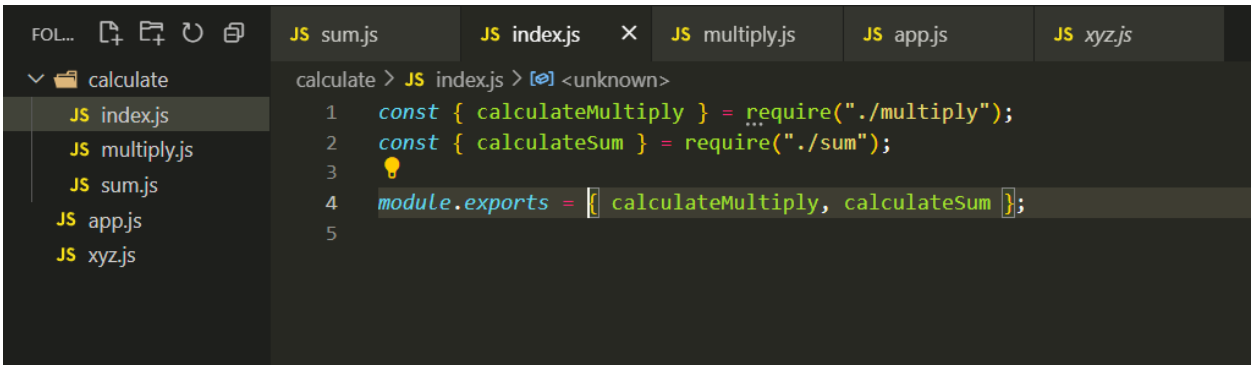
CREATE folder → calculate inside it create two files, **sum.js** and **multiply.js**



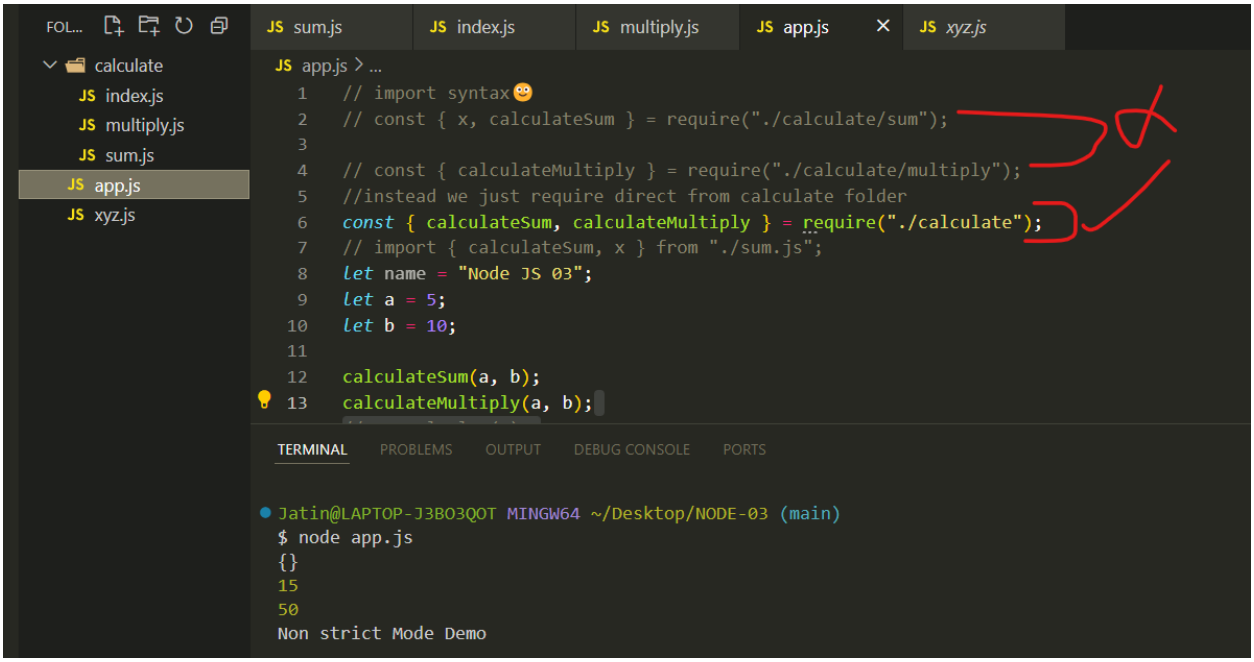


one more pattern

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



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

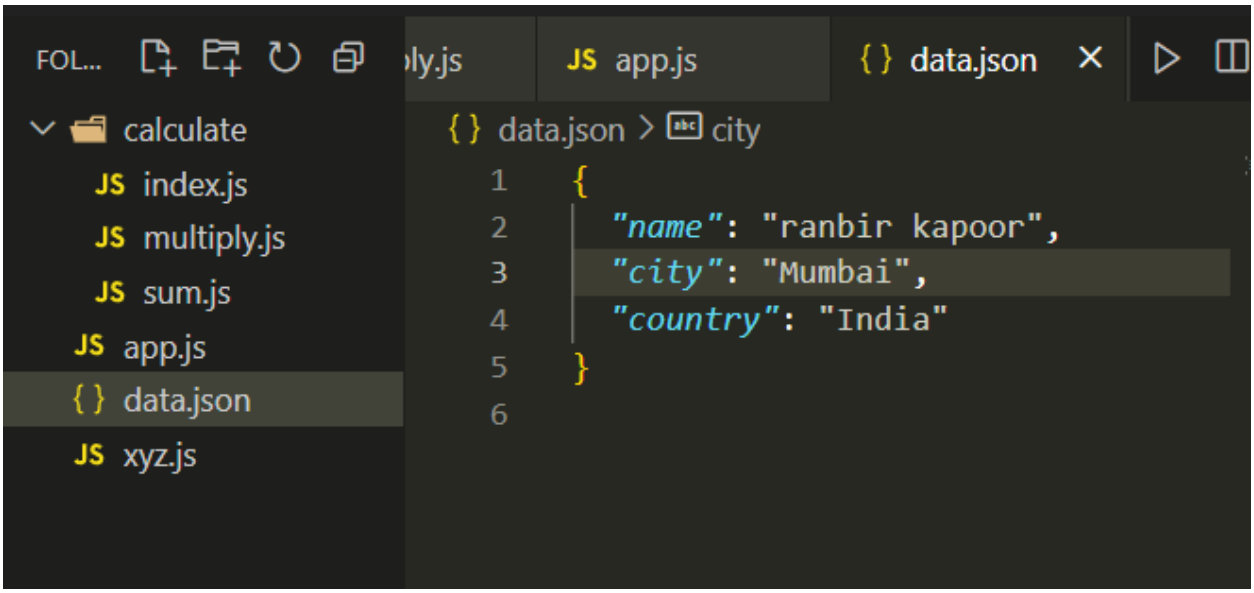


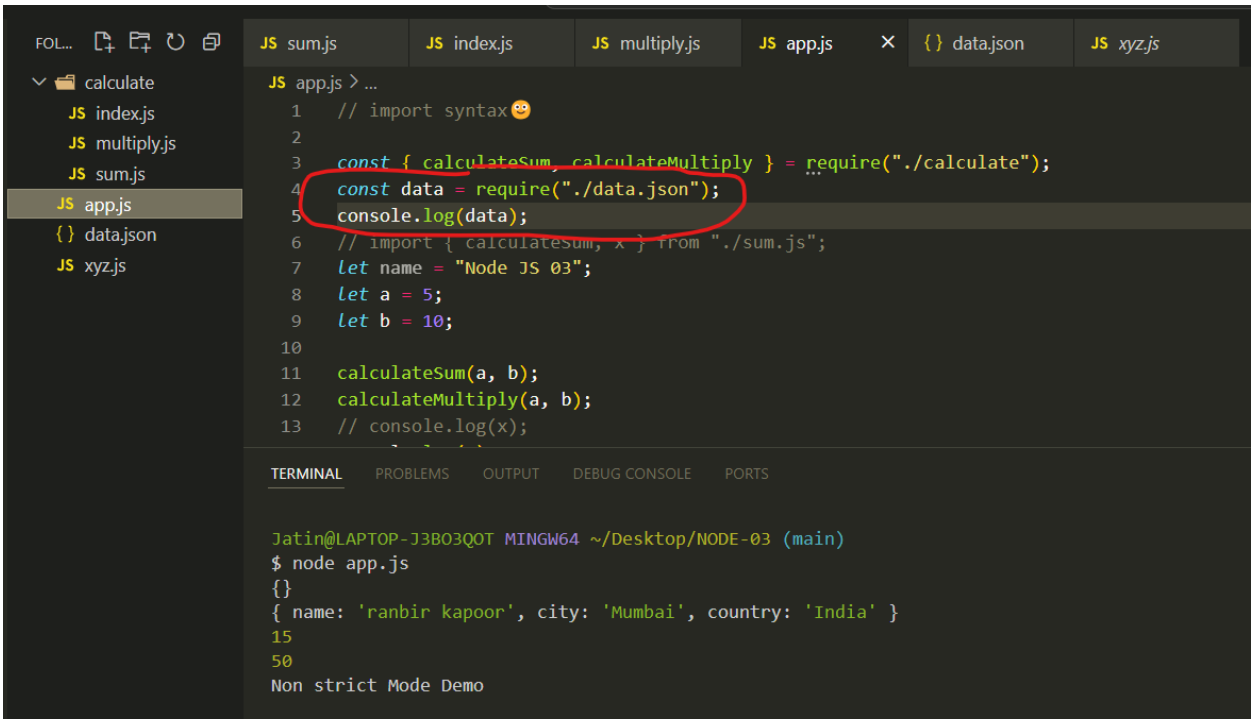
But why are we doing all this? When you have a large project with hundreds of files, it's beneficial to group related files together and create separate modules. In this case, the `calculate` folder serves as a new module that encapsulates related functionality, such as the `calculateSum` and `calculateMultiply`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





- 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