Episode-04 → module.exports & require

- we can't just keep writing all our code into a single file, actually it is possible we can just write all our Node.js code into a single file but logically we don't do it.

- When we create a project we create multiple files, multiple folders and then we can manage different type of files in a separate folder, there is a folder structure & directory structure involved.

- There are multiple files which are used to build our project, so how do we create those files and there is an important concept of modules.

- Whenever we have a Node.js application, there is one entry point in our application that entry point is a file that we give over terminal - node app.js.

- But what if there are code in someother file also, how will it be executed (apart from app.js, there is one more file xyz.js).

- Suppose if we wanted to execute xyz.js with app.js being the entry point, how do we execute that code.
  These two codes are not related, these two codes are very separate in some separate files.

- So, basically in Node.js we can call this xyz.js is a separate file/module.
  So how do we make two modules work together.

- The famous answer to this is a "require" function. There is a function which is just used to require other modules into our main module.

- We give require a path, now it's in the same directory, we can give "./".

- require ("./xyz.js"); in app.js, now if we run our file app.js we will see the execution of xyz.js.

- Basically whatever the code is there in xyz.js file that needs to be run first then the execution of app.js code be done.

- This is how we include or import a module inside another module.

- "require" function is available to us anywhere in our Node.js code, whenever we run any programme using Node.js "require" function is always available.
  Just like "Global" is there, similarly require is also there.

- We can just require any file or module inside another module, one module into another.

- app.js → require ("./xyz.js");

```
        Var name = "Namaste Node JS"
        Var  a  = 10;
        Var  b  = 20;
    console.log (globalThis === global);
```

- Suppose if we want to calculate the sum of 'a' and 'b'.

- We want to create a new module 'sum.js', it calculates the sum.

- sum.js →
```
console.log ("Sum Module Executed");
    function calculateSum (a,b) {
        const sum = a+b;
        console.log(sum);
    }
```

- Can we use this function inside our app.js file ?
```
        var a = 10;
        var b = 20;
        calculateSum (a, b);
```

- But it will not work like this, it will throw a Reference Error.

- after requiring sum.js it will also not work, require("./sum.js");

- Terminal —
```
• Very Important JS code
• Sum module executed
```

- But still the calculateSum is not available.

- whenever we create a separate module and we require that module this code will run but we can't access the variable, methods and functions of one module into another simply by requiring it.

- We can't directly access calculateSum( ) in app.js.

- Modules protect their Variables and function from leaking.

- Suppose if we want to give access of calculateSum to app.js, how would we do that, for that we will have to export this function from this file and then import that function into another file.

- If we want to give access calculateSum, so we have to explicitely export it.

- we use "module.exports = calculateSum";

- ~~Remember~~ Remember → It's exports, not export

- Over here it will still not work, because we will have to import that also.

  ex → const calculateSum = require ("./sum.js");

- so whatever we export from "module.exports" will be returned from the "require" function.

- const calculateSum = require ("./sum.js");

  Now, the exports calculateSum came over here in calculateSum variable.

- Now, if we run our programme, we will see the result of calculateSum in terminal window.

- Suppose if we had to export 'x' as well as 'calculateSum' function, how would we do that

Scanned with CamScanner

- sum.js →
```
console.log("Sum module Executed");
var x = "Hello world";
function calculateSum(a,b){
    const sum = a+b;
    console.log(sum);
}
```

- So, the way to export that is by wrapping it inside an object, we create a new object it has a method & method "x : x", calculateSum : calculateSum

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

- So, this same object is coming over in app.js.

- const calculateSum = require("./sum.js");
  (obj)

- const obj = require("./sum.js");

- Now we can do →

• obj.calculateSum(a,b)

• console.log(obj.x);

- Now we can export multiple things like this.

- In lot of places some people like to write like this -

```
const obj = require("./sum.js");
        ↓
const {x, calculateSum} = require("./sum.js");
```

They destructure it on the fly (Object destructuring).

- Now we don't need to write →
  - `obj.calculateSum(a, b);`
  - `console.log(obj.x);`

  Now, we will write -
  - `calculateSum(a, b);`
  - `console.log(x);`

-
```
module.exports = {
        x: x,
      calculateSum : calculateSum
  }
```

- This is an older way, we don't write the colon in front of it. Now JS itself will assume that 'x : x' is a shorthand.

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

- The important learning for us was that from a module we don't access it's private variable and functions outside unless the module wants it to be.

  This is a very powerful concept.

- Because the other modules have their own private space is a very big superpower it protects the variable.

- Suppose, in app.js we want to create a variable 'var x', so it will conflict with other modules, so to avoid that conflict, modules protect their private variables and functions.

- If we don't write '.js' extension in path name of 'require' function, it will be properly working, it is considered that we are using a '.js' extension.

→ we have seen the pattern of importing & exporting - 'require' & 'module.exports'.

  This type of pattern or module is known as 'Common JS modules' - CJS.

- There is one more thing which is known as ES Modules (it is also known as MJS)

- Just like we used 'module.exports' and then we use 'require()'. Similarly there is another pattern that is used is known as ES Modules.

- First of all we have to create a new file - package.json

- By default Common JS Module is enabled but if want to use ES Modules pattern, we have to write type in package.json.

```
- {
    "type" : "module"
  }
```

- This is a different way of importing & exporting modules, now all these modules are treated in a different way.

- In ES Modules (mjs system) we don't export like CJS, we write 'export' in front of function.

ex → Sum.js –

```
export function calculate$^{Sum}$(a, b) {
    const sum = a+b;
    console.log (sum);
}
```

app.js – `import {calculateSum} from './sum.js';`

- ES Modules by default used in React & Angular.
- We can do module export & import in EJS.

Sum.js –

```
export var x = "Hello world";
export function calculateSum (a, b) {
        const sum = a+b;
        console.log(sum);
```

app.js – `import {x, calculateSum} from './sum.js';`

- CJS is older way, ES Modules is a newer way.

- Open JS Foundation is now saying that going forward, ES Modules will be the standard way of importing & exporting modules.

- In this course we will be highly using CJS because rightnow almost all the repository of Node.js use this pattern and in Node.js this 'require()' and 'module.export' is a very big thing. In industry still we will find CJS module pattern being used.

- There is one major difference between CJS & EJS modules, when CJS is requiring these modules it does it in a synchronous way.

- Synchronous means the next line of code will only be executed once this require happen, basically it kind of like blocks for a while, until & unless 'sum.js' & 'xyz.js' is loaded in 'app.js' the code of 'app.js' will not move ahead.

  But in this way an option for async. This is very powerful, this is newer & better way of importing modules.

- In CJS the code run in non-strict mode and in ES Modules the code is run in strict mode.

- In strict mode we don't define variable without using var, let, const.

Q.> What is module.exports?

A.> module.exports is an empty object.

```
console.log(module.exports) → { }
```

- Instead of writing like this
  module.exports = {x, calculateSum};
  some people also prefer to write like this
  module.exports.x = x
  module.exports.calculateSum = calculateSum

- Earlier module.exports was an empty object now we are attaching these properties to same object.

- Suppose we want to nest modules (modules inside modules inside module)

- We make folder → Calculate
  ↓
  File → multiply.js
  → Sum.js

- Wherever we do 'module.exports' always wrap inside an object, it is very easy for us to understand. All this pattern is very good to follow.

- One more common pattern we will see when we create a folder inside a folder there is a collection of module also which can happen and one more pattern.

- Now we want to make 'calculate' folder as a module in itself.

- In 'Calculate' folder we will make one more file → index.js

- In this index.js we will kind of import (require) those two files - multiply.js, Sum.js.

index.js →

```
Const {CalculateMultiply} = require("./multiply);
Const {CalculateSum} = require("./sum");

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

— Now, in app.js →

Instead of writing these —

```
const {x, CalculateSum} = require ("./calculate/sum.js");
const {CalculateMultiply} = require ("./calculate/multiply.js");
```

we can write — index.js →

```
const {CalculateSum, CalculateMultiply} = require("./calculate");
```

app.js →

```
const {CalculateSum, CalculateMultiply} = require("./calculate/index")
```

we can also write without → '/index'

— when we have a lot of files, we basically try to group together these files and create a separate module out of it.

— "app.js" don't have to know how "calculate" folder structure it's file.

— "app.js" don't need to know that internally how does "calculate" distribute it's file.

— we have made a collection of "functions, variables" we have created multiple files, all these files are basically merging into index.js file

Scanned with CamScanner

and app.js is just talking to the 'Calculate' folder now, app.js doesn't even need to know the full file location.

- If we have a "data.json" file, how do we import this data from 'json' file.

- data.json →

```
{
  "name": "Rajesh Jha",
  "city" : "Faridabad,
  "country": "India"
}
```

- app.js →

```
const data = require("./data.json");
console.log(JSON.stringify(data));
```

↳ It's not mandatory

- There are some modules which are present inside the core of Node.js , there is a module which is known as util.

- app.js →

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

This util module gives the access to a util object and this util Object has a lot of important function and property.

For more PDF , github → rajeshjha2000