Modules in NodeJS

In Node.js, a module is essentially a reusable block of code that can be used to perform a specific set of tasks or provide a specific functionality. A module can contain variables, functions, classes, objects, or any other code that can be used to accomplish a particular task or set of tasks.

The primary purpose of using modules in Node.js is to help organize code into smaller, more manageable pieces. A modules can then be imported at any time and used flexibly which helps in creating reusable code components that can be shared across multiple projects.

To understand this, consider this example: Let's say you have defined lots of functions in your code that works with a huge volume of JSON data.

Losing your sleep and increased anxiety levels are common side effects of keeping all this stuff (functions + data + some other logic) in one single file.

So you, being a clever programmer, thought of making a separate file for the JSON data and a separate file for storing all the functions. Now, you can simply import the data and the functions whenever you want and use them accordingly. This method increases efficiency as your file size reduces drastically. This is the concept of modules!

Let's see how we can make our own modules. For this, we are going to write some code where we will be defining a

function called `sayHello()` in a file called `hello.js`. This function will accept a `name` as the parameter and simply print a greeting message in the console.

We will then import it in another file called `app.js` and use it there. How interesting, right 😂? Let's check out the code:

This is the code in `hello.js` file:

```
function  sayHello(name){

      console.log(`Hello ${name}`);

}



module.exports = sayHello
```

This is the code in `app.js` file:

```
const  sayHello = require('./hello.js');



sayHello('John');

sayHello('Peter');
```

```
sayHello('Rohit');
```

The file `hello.js` can be called the `module` in this case. Every module has an object called `exports` which should contain all the stuff you want to export from this module like variables or functions. In our case, we are defining a function in the `hello.js` file and directly exporting it.

The `app.js` file imports the `sayHello()` function from `hello.js` and stores it in the `sayHello` variable. To import something from a module, we use the `require()` method which accepts the path to the module. Now we can simply invoke the variable and pass a name as a parameter. Running the code in `app.js` file will produce the following output:

```
Hello John

Hello Peter

Hello Rohit
```

Short Note on `module.exports`

In the previous section of the article, we saw how to use `module.exports` but I felt that it is important to understand how it works in a bit more detail. Hence, this section of the article is like a mini tutorial where we will see how we can export

one variable/function as well as multiple variables and functions using `module.exports`. So, Let's get started:

`module.exports` is a special object in NodeJS that allows you to export functions, objects, or values from a module, so that other modules can access and use them. Here's an example of how to use `module.exports` to export a function from a module:

```
// myModule.js

function myFunction() {
  console.log('Hello from myFunction!');
}

module.exports = myFunction;
```

In this example, we define a function `myFunction` and then export it using `module.exports`. Other modules can now require this module and use the exported function:

```
// app.js

const myFunction = require('./myModule');

myFunction(); // logs 'Hello from myFunction!'
```

Everything seems fine now and life is good. But the problem arises when we have to export multiple functions and

variables from a single file. The point is when you use `module.exports` multiple times in a single module, it will replace the previously assigned value with the new one. Consider this code:

```
// module.js

function myFunction() {
  console.log('Hello from myFunction!');
}

function myFunction2() {
  console.log('Hello from myFunction2!');
}

// First Export
module.exports = myFunction;

// Second Export
module.exports = myFunction2;
```

In this example, we first export `myFunction()`. But we then overwrite `module.exports` with a new function - `myFunction2()`. As a result, only the second export statement will take effect, and the `myFunction()` function will not be exported.

This problem can be solved if you assign `module.exports` to an object which contains all the functions you want to export, like this:

```
// myModule.js

function myFunction1() {
  console.log('Hello from myFunction1!');
}

function myFunction2() {
  console.log('Hello from myFunction2!');
}

module.exports = {
  foo: 'bar',
  myFunction1: myFunction1,
  myFunction2: myFunction2
};
```

In this example, we export an object with three properties: `foo`, `myFunction1`, and `myFunction2`. Other modules can require this module and access these properties:

```
// app.js

const myModule = require('./myModule');

console.log(myModule.foo); // logs 'bar'
myModule.myFunction1(); // logs 'Hello from myFunction1!'
myModule.myFunction2(); // logs 'Hello from myFunction2!'
```

To summarize, you can use `module.exports` as many times as you want in your NodeJS code, but you should be aware that

each new assignment will replace the previous one. You should use an object to group multiple exports together.

Types Of Modules in Node

There are 2 types of modules in NodeJS:

- **Built In Modules**: These are modules included in Node by default, so you can use them without installation. You just need to import them and get started.
- **External Modules**: These are modules created by other developers which are not included by default. So you need to install them first before using them.