**02 Node Module System**

**1) Global Object**:

Node.js global objects are global in nature and they are available in all modules. We do not need to include these objects in our application, rather we can use them directly.

**Example**:

1. \_\_dirname
2. \_\_filename
3. Console
4. Process
5. Buffer
6. setImmediate(callback[, arg][, ...])
7. setInterval(callback, delay[, arg][, ...])
8. setTimeout(callback, delay[, arg][, ...])
9. clearImmediate(immediateObject)
10. clearInterval(intervalObject)
11. clearTimeout(timeoutObject)

**2) Modules**:

When we declare a function or variable that is added to the global scope may causes a problem. In real world application we are working with a lot of function, variable or file name. It is possible that we have two files or variable or functions have the same name.

As the function is define in the global scope, when we declare the function in another file the previous definition will be overwrite with the new definition. This is the problem in the global scope.

For resolve this type of problem we use module.

Module in Node.js is a simple or complex functionality organized in single or multiple JavaScript files which can be reused throughout the Node.js application.

Each module in Node.js has its own context, so it cannot interfere with other modules or pollute global scope. Also, each module can be placed in a separate .js file under a separate folder.

Every thing we declare in a module is private. That is, we cannot access it outside the module.

**Node.js Module Types**:

Node.js includes three types of modules

1. Core Modules
2. Local Modules
3. Third Party Modules

**Core Module**:

Node.js is a light weight framework. The core modules include bare minimum functionalities of Node.js. These core modules are compiled into its binary distribution and load automatically when Node.js process starts. However, we need to import the core module first in order to use it in our application.

For example, http, url, path, fs, util etc.

**Local Module**:

Local modules are modules created locally in your Node.js application. These modules include different functionalities of your application in separate files and folders. You can also package it and distribute it via NPM, so that Node.js community can use it. For example, if you need to connect to MongoDB and fetch data then you can create a module for it, which can be reused in your application.

**Third Party Modules**:

"Third-party" means any module not written by the developer own or included in the standard library. we can use them to add functionality to our code without having to write it ourselves.

**3) Creating a Modules**:

You can create your own modules, and easily include them in your applications.

Module is an object. We can see the details of a module.

console.log(module);

*/\**

*Module {*

*id: '.',*

*exports: {},*

*parent: null,*

*filename: 'C:\\Users\\ruhul\\Desktop\\node-practice\\app.js',*

*loaded: false,*

*children: [],*

*paths:*

*[ 'C:\\Users\\ruhul\\Desktop\\node-practice\\node\_modules',*

*'C:\\Users\\ruhul\\Desktop\\node\_modules',*

*'C:\\Users\\ruhul\\node\_modules',*

*'C:\\Users\\node\_modules',*

*'C:\\node\_modules' ] }*

*\*/*

Suppose we are creating a module name "logger.js" for login messages. We want to use the module in various part of our application.

**loger.js**:

*//suppose we provide service in online*

var url = "http://mylogger.io/log";

function log(message) {

*//sent an http request*

console.log(message);

}

Now the variable "url" and function "log" both scopes only in the logger file. They are private. We cannot access them outside the file. But we want to access log function from app.js file.

Now in the Module object one of the properties is "exports". The property is set to an empty object. Anything that we add to this object will be exported from this module and this is available outside of this module.

*//suppose we provide service in online*

var url = "http://mylogger.io/log";

function log(message) {

*//sent an http request*

console.log(message);

}

*//adding a method to export object and setting the log function to it*

module.exports.log = log;

We also can change the name of the property that we can access from outside.

module.exports.myLog = log;

Now the log method is public and accessible from outside.

**4) Loading a Modules**:

To lode a module we need the "require()" function. This function takes one argument, that’s the name or path of the target module we want to load. This function returns the object that is exported from the target module. We can store the object in a variable.

var logger = require("./logger");

Now we can call the log function from app.js file.

**app.js**:

*//since app.js and logger.js locket in the same folder*

var logger = require("./logger");

logger.log("Ruhul"); *//Ruhul*

The best practice is, when loading a module using the require() function its better to store the result in a constant. The reason is we don’t override the value of the variable.

const logger = require("./logger");

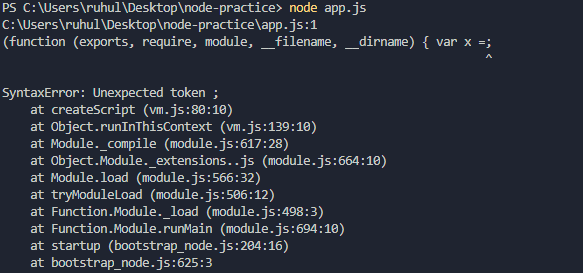
**5) Module Wrapper Function**:

The variable and functions define in a module are scoped that module. They are private and not visible from outside. Now we will show how node do this.

Write the following code in the first line of app.js file and run the file.

var x =;

We will get the following error.



We get an error "SyntaxError: Unexpected token ;". Above this line we see a function declaration.

(function (exports, require, module, \_\_filename, \_\_dirname)

This function has few parameters. exports, require, module, \_\_filename, and \_\_dirname. Which code we write in our program node don’t execute its directly. Internally node wraps it inside a function. At runtime our code converts as following

(function(exports, require, module, \_\_filename, \_\_dirname) {

var url = "http://mylogger.io/log";

function log(message) {

console.log(message);

}

module.exports.log = log;

});

We call this function as "Module Wrapper Function".

**Note**:

Actually, the function is much more complicated.

const logger = require("./logger");

logger.log("Ruhul"); *//Ruhul*

*/\**

*C:\Users\ruhul\Desktop\node-practice\logger.js*

*C:\Users\ruhul\Desktop\node-practice*

*Ruhul*

*\*/*

**6) Path Module**:

The path module provides utilities for working with file and directory paths.

const path = require("path");

var pathObj = path.parse(\_\_filename);

*//Formats a path string into a path object*

console.log(pathObj);

*/\**

*{ root: 'C:\\',*

*dir: 'C:\\Users\\ruhul\\Desktop\\node-practice',*

*base: 'app.js',*

*ext: '.js',*

*name: 'app' }*

*\*/*

**7) OS Module**:

The OS module provides a number of operating system-related utility methods. The syntax for including the OS module in your application.

var os = require('os');

**Example**:

const os = require("os");

var totalMemory = os.totalmem();

var freeMemory = os.freemem();

console.log(`Total memory: ${totalMemory}`);

console.log(`Free memory: ${freeMemory}`);

*/\**

*Total memory: 8481501184*

*Free memory: 4080324608*

*\*/*

**8) File System Module**:

The fs module provides an API for interacting with the file system in a manner closely modeled around standard POSIX functions. To use this module:

const fs = require('fs');

**POSIX**:

The Portable Operating System Interface (POSIX) is a family of standards specified by the IEEE Computer Society for maintaining compatibility between operating systems. POSIX defines the application programming interface (API), along with command line shells and utility interfaces, for software compatibility with variants of Unix and other operating systems.

Common use for the File System module is:

1. Read files
2. Create files
3. Update files
4. Delete files
5. Rename files

**Example**:

const fs = require("fs");

*//returens all the files and folder present in current folder*

const files = fs.readdirSync("./");

console.log(files);

*/\**

*[ 'app.js', 'logger.js' ]*

*\*/*

**Example**:

Asynchronous from

const fs = require("fs");

const files = fs.readdir("./", function(err, files) {

if (err) {

console.log("Error", err);

} else {

console.log("Result", files);

}

});

*/\**

*Result [ 'app.js', 'logger.js' ]*

*\*/*

**Error**:

const fs = require("fs");

const files = fs.readdir("$", function(err, files) {

if (err) {

console.log("Error", err);

} else {

console.log("Result", files);

}

});

*/\**

*Error { Error: ENOENT: no such file or directory, scandir 'C:\Users\ruhul\Desktop\node-practice\$'*

*errno: -4058,*

*code: 'ENOENT',*

*syscall: 'scandir',*

*path: 'C:\\Users\\ruhul\\Desktop\\node-practice\\$' }*

*\*/*

**Note**:

All the methods present in " File System" module is Synchronous and Asynchronous from.

**9) Event Module**:

One of the core concepts in node is the concept of events. A lot of node core functionality is based on the concept of Events. An event is basically a signal that something has happened in our application.

The Events module provides a way of working with events. In Node.js, all events are an instance of the EventEmitter object

**Example**:

*//indicate that EventEmitter is a class*

const EventEmitter = require("events");

*//emitter is an object*

const emitter = new EventEmitter();

*//emit means making a noise or produce something*

*//here signaling in our application that something happened*

emitter.emit("messageLogged");

If we run this application noting happened. Because we have raised an event here but nowhere in our application register a listener that is interested to the event. A listener is a function that will be called when a particular event is raised.

**Example**:

const EventEmitter = require("events");

const emitter = new EventEmitter();

*//Register a listener*

emitter.on("messageLogged", function() {

console.log("Listner called");

});

*//Raise an event*

emitter.emit("messageLogged");

*/\**

*Listner called*

*\*/*

Here, when we raise this event, callback function or listener is called.

**Note**:

The order is important here. If we register the listener after calling the emit() method, nothing be happened. Because when we call the emit method this emitter iterates over all the registered listener and calls them synchronously.

**10) Event Arguments**:

When we are rise an event, we also can send some data about that event. For example, in our logger module when we log a message par haves from a remote login service will generates an id for that message and returns the id to the client. Or it may give us a URL to access that logged message directly.

So, when raising an event, we can add additional arguments which refer to event arguments.

*//Raise an event*

*//1=> id, url=> url*

emitter.emit("messageLogged", 1, url);

But this value of event argument is little bit confusing. So, if we want to sent multiple values about an event, it’s a better practice to encapsulates those values into an object.

*//Raise an event*

*//{ id: 1, url: "http://" } => object with multiple value*

emitter.emit("messageLogged", { id: 1, url: "http://" });

Now when registering this listener, the caller function of the listener can also receive the event arguments.

*//Register a listener*

*//we can add any name as argument but the convention is args or e or eventArg*

emitter.on("messageLogged", function(args) {

console.log("Listner called", args);

});

**Example**:

const EventEmitter = require("events");

const emitter = new EventEmitter();

*//Register a listener*

*//we can add any name as argument but the cinvention is args or e or eventArg*

emitter.on("messageLogged", function(args) {

console.log("Listner called", args);

});

*//Raise an event*

*//{ id: 1, url: "http://" } => object with multiple value*

emitter.emit("messageLogged", { id: 1, url: "http://" });

*/\**

*Listener called { id: 1, url: 'http://' }*

*\*/*

By following this technique, we can also pass data about the event what just happened. Now we make this code little bit simple. In ES6 we have the feature called arrow function. We can use arrow function to make our code little bit simple.

emitter.on("messageLogged", args => {

console.log("Listner called", args);

});

*/\**

*Listener called { id: 1, url: 'http://' }*

*\*/*

**11) Extending Event Emitter**:

In the real-world application it is very rower that we are working with the EventEmiter object directly.

const emitter = new EventEmitter();

Instead, we are creating a separate class that has all the capabilities of the EVentEmiter and then use that class in our code.

**Logger.js**:

const EventEmitter = require("events");

const emitter = new EventEmitter();

var url = "http://mylogger.io/log";

function log(message) {

console.log(message);

*//Raisee an event*

emitter.emit("messageLogged", { id: 1, url: "http://" });

}

module.exports = log;

**app.js**:

const EventEmitter = require("events");

const emitter = new EventEmitter();

emitter.on("messageLogged", args => {

console.log("Listner called", args);

});

const log = require("./logger");

log("message"); *//message*

Now if we run "app.js" we only see the output "message". Our event listener was not called. The reason for this is because here we are working with two different EventEmitter. In app.js and logger.js we have two EventEmitter.

const emitter = new EventEmitter();

In this case we have two different objects. In logger module we use an emitter to emit an event and in app.js module we have also an emitter to emit an event. Those two events are completely different.

When we register an event in a module that listener is only registered with this EventEmitter. And this is completely different from another module.

Here we create a class call Logger that have additional method log(). And from logger.js we are export the Logger class. To add the EventEmitter related functionality we have to extends the EventEmitter class in our Logger class. And remove the following line

const emitter = new EventEmitter();

**Logger.js**:

const EventEmitter = require("events");

var url = "http://mylogger.io/log";

class Logger extends EventEmitter {

log(message) {

console.log(message);

*//Raisee an event*

*this*.emit("messageLogged", { id: 1, url: "http://" });

}

}

module.exports = Logger;

**app.js**:

const Logger = require("./logger");

const logger = new Logger();

logger.on("messageLogged", args => {

console.log("Listener called", args);

});

logger.log("message");

*/\**

*message*

*Listener called { id: 1, url: 'http://' }*

*\*/*

**12) HTTP Module**:

One of the powerful building blocks of node is the http module that we use to creating networking applications. Node.js has a built-in module called HTTP, which allows Node.js to transfer data over the Hyper Text Transfer Protocol (HTTP). To include the HTTP module, use the require() method:

var http = require('http');

**Example**:

const http = require("http");

const server = http.createServer();

server.on("connection", socket => {

console.log("New Connection...");

});

server.listen(3000);

console.log("Listening on port 3000...");

Now if we run app.js we will see the following output in the console.

Listening on port 3000...

Now go to the browser and in address bar write "localhost:3000", we will see the following output.

Listening on port 3000...

New Connection...

New Connection...

But in real world application we are not going to response to the connection event to build an http service. This is very low level. We pass a callback function to the " createServer()" method.

**App.js**:

const http = require("http");

const server = http.createServer((req, res) => {

if (req.url == "/") {

res.write("Hello World");

res.end();

}

if (req.url == "/api/courses") {

res.write(*JSON*.stringify([1, 2, 3, 4]));

res.end();

}

});

server.listen(3000);

console.log("Listening on port 3000...");

Now if we run app.js we will see the following output in the console.

Listening on port 3000...

Now go to the browser and in address bar write "localhost:3000/api/courses", we will see the following output.

[1,2,3,4]

In read world we are not going to use this http module to build in a backend service for our application. The reason for this is because as we can see here, as we add more routes this code more complex, because we add all of them in Linear way inside a callback function.

Instead we use framework called express which gives our application a clean structure to handle various route. Internally the express framework is build on top of the http module in node.

02 Node Module System