**Node.js**:It follows Single Threaded with Event Loop Model. Node JS Processing model mainly based on Javascript Event based model with Javascript callback mechanism.

**Single Threaded Event Loop Model Processing Steps**:

* Clients Send request to Web Server.
* Node JS Web Server internally maintains a Limited Thread pool to provide services to the Client Requests.
* Node JS Web Server receives those requests and places them into a Queue. It is known as “Event Queue”.
* Node JS Web Server internally has a Component, known as “Event Loop”. Why it got this name is that it uses indefinite loop to receive requests and process them. (See some Java Pseudo code to understand this below).
* Event Loop uses Single Thread only. It is main heart of Node JS Platform Processing Model.
* Even Loop checks any Client Request is placed in Event Queue. If no, then wait for incoming requests for indefinitely.
* If yes, then pick up one Client Request from Event Queue
  + Starts process that Client Request
  + If that Client Request Does Not requires any Blocking IO Operations, then process everything, prepare response and send it back to client.
  + If that Client Request requires some Blocking IO Operations like interacting with Database, File System, External Services then it will follow different approach
    - Checks Threads availability from Internal Thread Pool
    - Picks up one Thread and assign this Client Request to that thread.
    - That Thread is responsible for taking that request, process it, perform Blocking IO operations, prepare response and send it back to the Event Loop
    - **Event Loop in turn, sends that Response to the respective Client.**

**EventEmitter** class contains all required functions to take care of generating events. EventEmitter class is responsible to generate events. Generating events is also known as Emitting. That’s why this class name is EventEmitter as it emits events in Node JS Platform.



* When Node JS application starts or ends an operation, EventEmitter class generates events and places them into Event Queue.
* Event Queue maintains a Queue of Events.
* Event Loop continuously waits for new events in Event Queue. When it finds events in Event Queue, it pulls them and try to process them. If they require IO Blocking operations or long waiting tasks, then assigns respective Event Handlers to handle them.
* Event Handlers are JavaScript Asynchronous Callback Functions. They are responsible to handle events and return results to Event Loop.
* Event Loop will prepare results and send them back to the Client.

Sync vs async:- When you execute something synchronously, you wait for it to finish before moving on to another task. When you execute something asynchronously, you can move on to another task before it finishes.

var events = require("events");

var eventsEmitter = new events.EventEmitter();

eventsEmitter.emit("mobileon",mobileOnHadler);

eventsEmitter.emit("mobileon");

function mobileOnHadler(data){

    console.log(data);

}

**Callback function** is used in node.js to deal with multiple requests made to the server. Like if you have a large file which is going to take a long time for a server to read and if you don’t want a server to get engage in reading that large file while dealing with other requests, call back function is used. Call back function allows the server to deal with pending request first and call a function when it is finished.

A **callback function**, also known as a higher-order **function**, is a **function** that is passed to another **function** (let's call this other **function** “otherFunction”) as a parameter, and the **callback function** is called (or executed) inside the otherFunction.

**Callback hell:-**It is nothing **but** writing callback function inside callback.

To avoid callback hell you can use:-

**Keep your code shallow:- var** form **=** document.querySelector('form')

form.onsubmit **=** **function** **formSubmit** (submitEvent) {

**var** name **=** document.querySelector('input').value

request({

uri: "http://example.com/upload",

body: name,

method: "POST"

}, **function** **postResponse** (err, response, body) {

**var** statusMessage **=** document.querySelector('.status')

**if** (err) **return** statusMessage.value **=** err

statusMessage.value **=** body

})

}

document.querySelector('form').onsubmit **=** formSubmit

**function** **formSubmit** (submitEvent) {

**var** name **=** document.querySelector('input').value

request({

uri: "http://example.com/upload",

body: name,

method: "POST"

}, postResponse)

}

**function** **postResponse** (err, response, body) {

**var** statusMessage **=** document.querySelector('.status')

**if** (err) **return** statusMessage.value **=** err

statusMessage.value **=** body

}

## Modularize

module.exports.submit **=** formSubmit

**function** **formSubmit** (submitEvent) {

**var** name **=** document.querySelector('input').value

request({

uri: "http://example.com/upload",

body: name,

method: "POST"

}, postResponse)

}

**function** **postResponse** (err, response, body) {

**var** statusMessage **=** document.querySelector('.status')

**if** (err) **return** statusMessage.value **=** err

statusMessage.value **=** body

}

### Async.js

var fs = require('fs');

var async = require('async');

var myFile = '/tmp/test';

async.waterfall([

function(callback) {

fs.readFile(myFile, 'utf8', callback);

},

function(txt, callback) {

txt = txt + '\nAppended something!';

fs.writeFile(myFile, txt, callback);

}

], function (err, result) {

if(err) return console.log(err);

console.log('Appended text!');

});

### Promises

var Promise = require('bluebird');

var fs = require('fs');

Promise.promisifyAll(fs);

var myFile = '/tmp/test';

fs.readFileAsync(myFile, 'utf8').then(function(txt) {

txt = txt + '\nAppended something!';

fs.writeFile(myFile, txt);

}).then(function() {

console.log('Appended text!');

}).catch(function(err) {

console.log(err);

});

### Async/Await

async function getUser(id) {

if (id) {

return await db.user.byId(id);

} else {

throw 'Invalid ID!';

}

}

try {

let user = await getUser(123);

} catch(err) {

console.error(err);

}

Link for Callback hell -http://blog.vullum.io/javascript-flow-callback-hell-vs-async-vs-highland/

Global dependency:-npm install –g pkgname

**Npm-init:-**That way when we run npm init, it will already know what repository you're pulling from and your git remote will be properly setup to track origin/master.

This will initate a command line questionnaire that will conclude with the creation of a**package.json** in the directory you initiated the command.

## [Updating npm](https://docs.npmjs.com/getting-started/installing-node#updating-npm):- npm install npm –g

## How to connect client to server.(using http,socket.io).

## *Scalability* is the capability of a system, network, or process to handle a growing amount of work

***Middleware*** functions are functions that have access to the [request object](https://expressjs.com/en/4x/api.html#req) (req), the [response object](https://expressjs.com/en/4x/api.html#res) (res), and the next middleware function in the application’s request-response cycle. The next middleware function is commonly denoted by a variable named next.

Middleware functions can perform the following tasks:

* Execute any code.
* Make changes to the request and the response objects.
* End the request-response cycle.
* Call the next middleware in the stack.

## Routing refers to the definition of application end points (URIs) and how they respond to client requests

The core idea behind promises is that a promise represents the result of an asynchronous operation. A promise is in one of three different states:

* pending - The initial state of a promise.
* fulfilled - The state of a promise representing a successful operation.
* rejected - The state of a promise representing a failed operation.

Once a promise is fulfilled or rejected, it is immutable (i.e. it can never change again).

**Module.exports and exports**

|  |
| --- |
| 'use strict';  module.exports = function(name, age) {       return {     get\_name: function() {return name},     get\_age: function() {return age}       }  } |

And, main.js would look like this

|  |  |
| --- | --- |
|  | 'use strict';  var Person = require('./person');  var person = new Person('Bibek', 25);  console.log(person.get\_name()); |

Running main.js would give name of the person.  
  
So far I showed you way of importing file into another file using module.exports. But what if I replacemodule.exports with exports in above code? ... this wont work. This is really confusing for beginner.  
  
To illustrate why replacing module.exports with exports doesn't work, let me revise your basic Javascript knowledge.

|  |
| --- |
| exports = function() {   console.log('with exports only');  } |

when we assign function to exports. Following will be the result of exports and module.exports

exports = [function],

module.exports = {}

Since module.exports will be returned, this will return empty object. So this doesn't work.  
  
Now, consider example 2

|  |  |
| --- | --- |
| 1  2  3 | exports.a = function () {    console.log('exports with mutation');  } |

When we mutate exports, following will be the result of exports and module.exports

exports = {a: [function]},

module.exports = {a: [funciton]}

This will return non-empty object. So this works  
Finally, consider example 3

|  |  |
| --- | --- |
| 1  2  3 | module.exports = function() {      console.log('with  module.exports only');  } |

When we assign function to module.exports, following will be the contents of exports and  
module.exports

exports = {},

This will return non-empty object. So this also works.

**What is the difference between exports and module.exports in Node.js?**

You must be familiar with the exports object in Node.js modules, using which you create functions in your modules like this (assume in a file named rocker.js):

exports.name = function() {  
 console.log('My name is Lemmy Kilmister');  
};

which you call from another file thus:

var rocker = require('./rocker.js');  
rocker.name(); // 'My name is Lemmy Kilmister'

But what the heck is module.exports? Is it even legal?

Here is an eye-opener - **module.exports is the real deal**. exports is just module.exports's little helper. Your module returns module.exports to the caller ultimately, not exports. All exports does is collect properties and attach them to module.exports IF module.exports doesn't have something on it already. If there's something attached to module.exports already, everything on exports is ignored.

Put the following in rocker.js:

module.exports = 'ROCK IT!';  
exports.name = function() {  
 console.log('My name is Lemmy Kilmister');  
};

And this in another file, and run it:

var rocker = require('./rocker.js');  
rocker.name(); // TypeError: Object ROCK IT! has no method 'name'

The rocker module completely ignored exports.name, and returned a string 'ROCK IT!'. From that you probably realize that your modules don't always have to be 'module instances'. Your modules can be any legal JavaScript object - boolean, number, date, JSON, string, function, array, and so on. Your module is whatever you set module.exports to. If you don't set module.exports to anything explicitly, the properties of exports and attached to it and returned.

In this case, your module is a class:

module.exports = function(name, age) {  
 this.name = name;  
 this.age = age;  
 this.about = function() {  
 console.log(this.name +' is '+ this.age +' years old');  
 };  
};

and you'd use it this way:

var Rocker = require('./rocker.js');  
var r = new Rocker('Ozzy', 62);  
r.about(); // Ozzy is 62 years old

In this case, your module is an array:

module.exports = ['Lemmy Kilmister', 'Ozzy Osbourne', 'Ronnie James Dio', 'Steven Tyler', 'Mick Jagger'];

and you may use it this way:

var rocker = require('./rocker.js');  
console.log('Rockin in heaven: ' + rocker[2]); //Rockin in heaven: Ronnie James Dio

So you get the point now - if you want your module to be of a specific object type, usemodule.exports; if you want your module to be a typical module instance, use exports.

The result of attaching properties to module.exports is akin to attaching properties to exports. For example this:

module.exports.name = function() {  
 console.log('My name is Lemmy Kilmister');  
};

does the same thing as:

exports.name = function() {  
 console.log('My name is Lemmy Kilmister');  
};

But note that, they are not the same thing. As I said earlier module.exports is the real deal, exportsis just its little helper. Having said that, exports is the recommended object unless you are planning to change the object type of your module from the traditional 'module instance' to something else.

I hope this post helped you understand the difference between exports and module.exports, and learn a bit more about how modules work in Node.js. Any questions, ping me in the comments.

UPDATE: 7th Feb, 2014

As long are you don't overwrite the module.exports object with an assignment operation, anything attached to module.exports and exports will be available in the 'required' module.

If this is the content of your module:

module.exports.age = 68;  
exports.name = 'Lemmy Kilmister';

The following code would work fine:

var rocker = require('./rocker.js');  
console.log('%s is %s', rocker.name, rocker.age); // Lemmy Kilmister is 68

BUT

if you overwrite module.exports with anything in your module, it will fail:

module.exports = 'LOL';  
module.exports.age = 68;  
exports.name = 'Lemmy Kilmister';

or

module.exports.age = 68;  
exports.name = 'Lemmy Kilmister';  
module.exports = 'WTF';

the order doesn't matter, rocker.age and rocker.name will now be undefined.

Also, note: just because module.exports.age and exports.name are exported, does not mean you should use a combination of both. My recommendation is to stick to exports.\*, and be aware ofmodule.exports.\*.

**Type of javascript:-**

* Six data types that are [primitives](https://developer.mozilla.org/en-US/docs/Glossary/Primitive):
  + [Boolean](https://developer.mozilla.org/en-US/docs/Glossary/Boolean)
  + [Null](https://developer.mozilla.org/en-US/docs/Glossary/Null)
  + [Undefined](https://developer.mozilla.org/en-US/docs/Glossary/Undefined)
  + [Number](https://developer.mozilla.org/en-US/docs/Glossary/Number)
  + [String](https://developer.mozilla.org/en-US/docs/Glossary/String)
  + [Symbol](https://developer.mozilla.org/en-US/docs/Glossary/Symbol) (new in ECMAScript 6)
* and [Object](https://developer.mozilla.org/en-US/docs/Glossary/Object)

Add two number :- function makeAdder(x) {

return function(y) {

return x + y;

};

}

var add5 = makeAdder(5);

var add10 = makeAdder(10);

console.log(add5(2)); // 7

console.log(add10(2)); // 12

**Closures are functions that refer to independent (free) variables (variables that are used locally, but defined in an enclosing scope). In other words, these functions 'remember' the environment in which they were created.**

function init() {

var name = "Mozilla"; // name is a local variable created by init

function displayName() { // displayName() is the inner function, a closure

alert(name); // use variable declared in the parent function

}

displayName();

}

init();

 it will negatively affect script performance both in terms of processing speed and memory consumption.

Hoisting:- a variable can be used before it has been declared.-----

bla = 2 ;var bla;

<https://scotch.io/tutorials/understanding-hoisting-in-javascript>

<https://stackoverflow.com/questions/16383795/difference-between-module-exports-and-exports-in-the-commonjs-module-system>

<http://exploringjs.com/es6/ch_core-features.html#sec_from-var-to-const>

<https://www.tutorialspoint.com/nodejs/nodejs_mock_test.htm>

<https://scotch.io/tutorials/nodejs-tests-mocking-http-requests>

<http://vansande.org/2015/03/22/unit_testing_with_mocks_in_node_js/>

A closure is basically just a different way of looking at an object. An object is data that has one or more functions bound to it. A closure is a function that has one or more variables bound to it. The two are basically identical, at an implementation level at least. The real difference is in where they come from.