Node.js Tutorial

Node.js tutorial provides basic and advanced concepts of Node.js. Our Node.js tutorial is designed for beginners and professionals both.

Node.js is a cross-platform environment and library for running JavaScript applications which is used to create networking and server-side applications.

Our Node.js tutorial includes all topics of Node.js such as Node.js installation on windows and linux, REPL, package manager, callbacks, event loop, os, path, query string, cryptography, debugger, URL, DNS, Net, UDP, process, child processes, buffers, streams, file systems, global objects, web modules etc. There are also given Node.js interview questions to help you better understand the Node.js technology.

What is Node.js

Node.js is a cross-platform runtime environment and library for running JavaScript applications outside the browser. It is used for creating server-side and networking web applications. It is open source and free to use. It can be downloaded from this link <https://nodejs.org/en/>

Many of the basic modules of Node.js are written in JavaScript. Node.js is mostly used to run real-time server applications.

The definition given by its official documentation is as follows:

?Node.js is a platform built on Chrome's JavaScript runtime for easily building fast and scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.?

Node.js also provides a rich library of various JavaScript modules to simplify the development of web applications.

1. Node.js = Runtime Environment + JavaScript Library

**Different parts of Node.js**

The following diagram specifies some important parts of Node.js:

Features of Node.js

Following is a list of some important features of Node.js that makes it the first choice of software architects.

1. **Extremely fast:**Node.js is built on Google Chrome's V8 JavaScript Engine, so its library is very fast in code execution.
2. **I/O is Asynchronous and Event Driven:**All APIs of Node.js library are asynchronous i.e. non-blocking. So a Node.js based server never waits for an API to return data. The server moves to the next API after calling it and a notification mechanism of Events of Node.js helps the server to get a response from the previous API call. It is also a reason that it is very fast.
3. **Single threaded:**Node.js follows a single threaded model with event looping.
4. **Highly Scalable:**Node.js is highly scalable because event mechanism helps the server to respond in a non-blocking way.
5. **No buffering:**Node.js cuts down the overall processing time while uploading audio and video files. Node.js applications never buffer any data. These applications simply output the data in chunks.
6. **Open source:**Node.js has an open source community which has produced many excellent modules to add additional capabilities to Node.js applications.
7. **License:**Node.js is released under the MIT license.

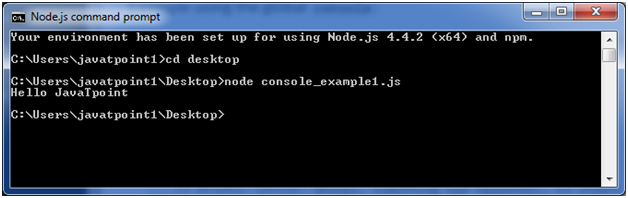
Node.js console-based Example

*File: console\_example1.js*

1. console.log('Hello JavaTpoint');

Open Node.js command prompt and run the following code:

1. node console\_example1.js



Here, console.log() function displays message on console.

Node.js web-based Example

A node.js web application contains the following three parts:

1. **Import required modules:** The "require" directive is used to load a Node.js module.
2. **Create server:**You have to establish a server which will listen to client's request similar to Apache HTTP Server.
3. **Read request and return response:** Server created in the second step will read HTTP request made by client which can be a browser or console and return the response.

**How to create node.js web applications**

Follow these steps:

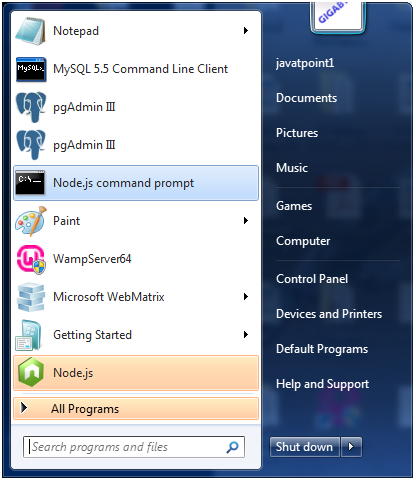
1. **Import required module:**The first step is to use ?require? directive to load http module and store returned HTTP instance into http variable. For example:
   1. var http = require("http");
2. **Create server:**In the second step, you have to use created http instance and call http.createServer() method to create server instance and then bind it at port 8081 using listen method associated with server instance. Pass it a function with request and response parameters and write the sample implementation to return "Hello World". For example:
   1. http.createServer(function (request, response) {
   2. // Send the HTTP header
   3. // HTTP Status: 200 : OK
   4. // Content Type: text/plain
   5. response.writeHead(200, {'Content-Type': 'text/plain'});
   6. // Send the response body as "Hello World"
   7. response.end('Hello World\n');
   8. }).listen(8081);
   9. // Console will print the message
   10. console.log('Server running at http://127.0.0.1:8081/');
3. **Combine step1 and step2 together** in a file named "main.js".

*File: main.js*

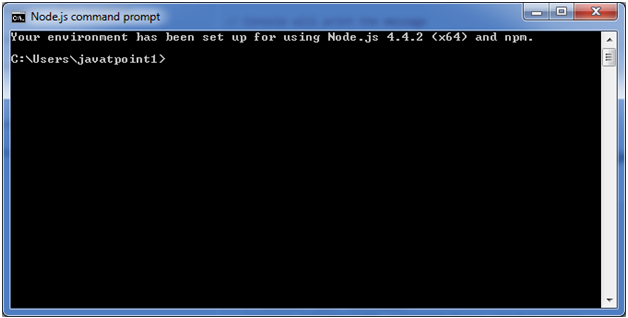
* 1. var http = require("http");
  2. http.createServer(function (request, response) {
  3. // Send the HTTP header
  4. // HTTP Status: 200 : OK
  5. // Content Type: text/plain
  6. response.writeHead(200, {'Content-Type': 'text/plain'});
  7. // Send the response body as "Hello World"
  8. response.end('Hello World\n');
  9. }).listen(8081);
  10. // Console will print the message
  11. console.log('Server running at http://127.0.0.1:8081/');

**How to start your server:**

Go to start menu and click on the Node.js command prompt.



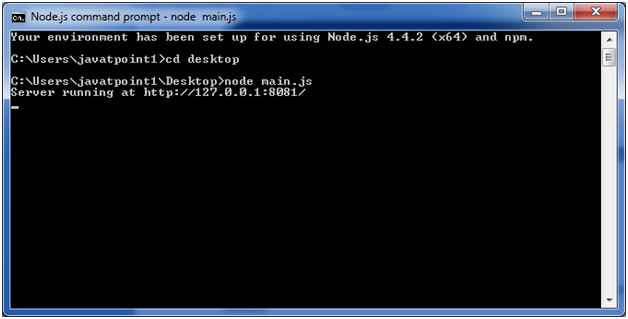
Now command prompt is open:



**Set path:**Here we have save "main.js" file on the desktop.

So type **cd desktop** on the command prompt. After that execute the main.js to start the server as follows:

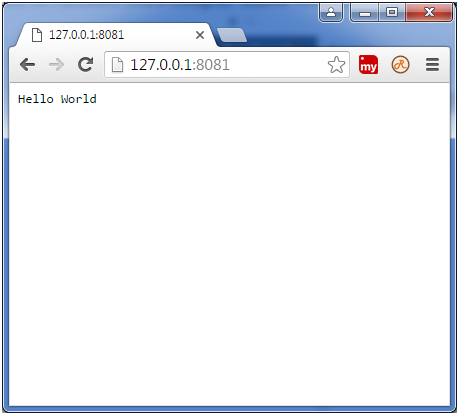
1. node main.js



Now server is started.

**Make a request to Node.js server:**

Open http://127.0.0.1:8081/ in any browser. You will see the following result.



Now, if you make any changes in the "main.js" file, you need to again run the "node main.js" command.

Node.js Console

The Node.js console module provides a simple debugging console similar to JavaScript console mechanism provided by web browsers.

There are three console methods that are used to write any node.js stream:

1. console.log()
2. console.error()
3. console.warn()

Node.js console.log()

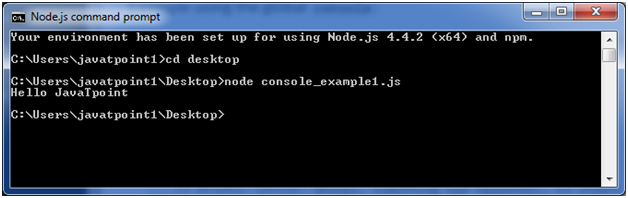
The console.log() function is used to display simple message on console.

*File: console\_example1.js*

1. console.log('Hello JavaTpoint');

Open Node.js command prompt and run the following code:

1. node console\_example1.js



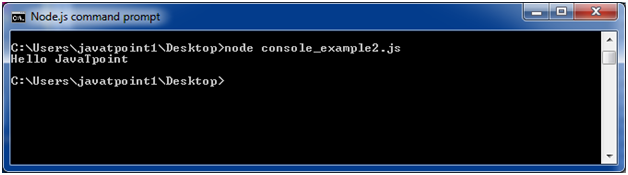
We can also use format specifier in console.log() function.

*File: console\_example2.js*

1. console.log('Hello %s', 'JavaTpoint');

Open Node.js command prompt and run the following code:

1. node console\_example2.js



Node.js console.error()

The console.error() function is used to render error message on console.

*File: console\_example3.js*

1. console.error(**new** Error('Hell! This is a wrong method.'));

Open Node.js command prompt and run the following code:

1. node console\_example3.js



Node.js console.warn()

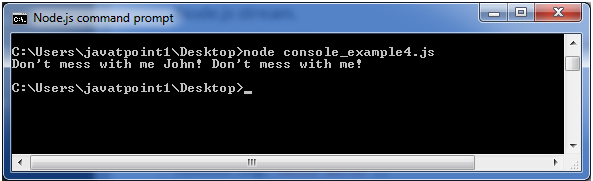
The console.warn() function is used to display warning message on console.

*File: console\_example4.js*

1. **const** name = 'John';
2. console.warn(`Don't mess with me ${name}! Don't mess with me!`);

Open Node.js command prompt and run the following code:

1. node console\_example4.js



Node.js REPL

The term REPL stands for **Read Eval Print**and**Loop**. It specifies a computer environment like a window console or a Unix/Linux shell where you can enter the commands and the system responds with an output in an interactive mode.

REPL Environment

The Node.js or node come bundled with REPL environment. Each part of the REPL environment has a specific work.

**Read:** It reads user's input; parse the input into JavaScript data-structure and stores in memory.

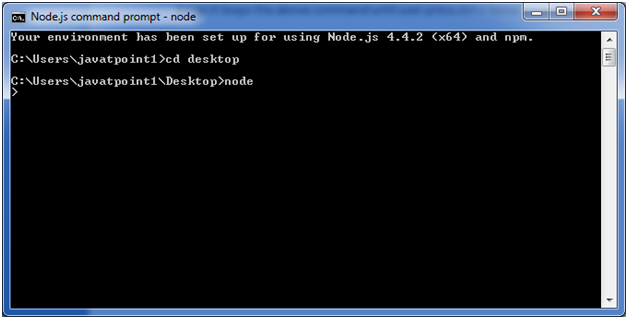
**Eval:**It takes and evaluates the data structure.

**Print:**It prints the result.

**Loop:** It loops the above command until user press ctrl-c twice.

How to start REPL

You can start REPL by simply running "node" on the command prompt. See this:

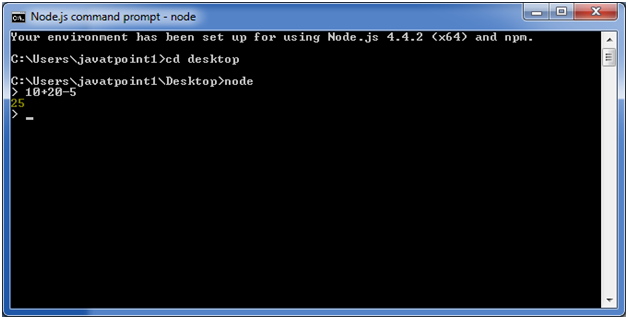


You can execute various mathematical operations on REPL Node.js command prompt:

Node.js Simple expressions

After starting REPL node command prompt put any mathematical expression:

1. Example: **>**10+20-5
2. 25



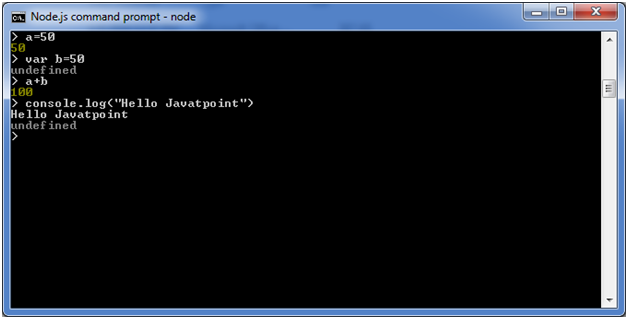
1. Example2: **>**10+12 + (5\*4)/7



Using variable

Variables are used to store values and print later. If you don't use **var**keyword then value is stored in the variable and printed whereas if **var** keyword is used then value is stored but not printed. You can print variables using console.log().

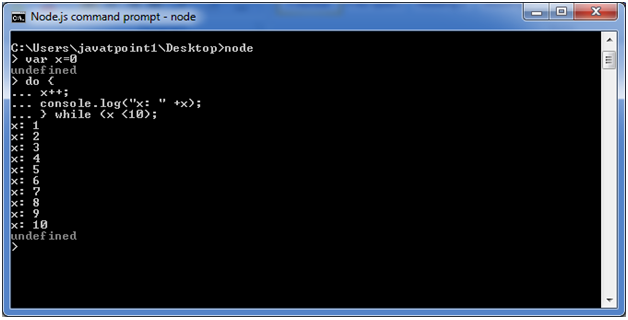
**Example:**



Node.js Multiline expressions

Node REPL supports multiline expressions like JavaScript. See the following do-while loop example:

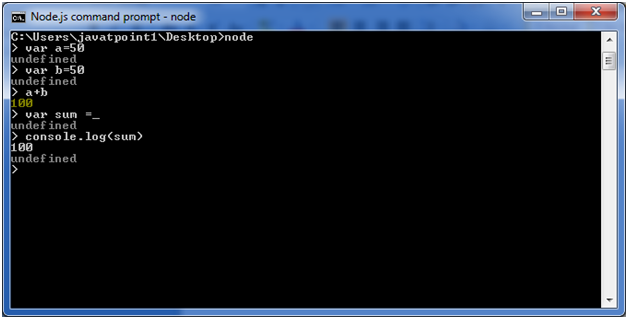
1. var x = 0
2. undefined
3. **>** do {
4. ... x++;
5. ... console.log("x: " + x);
6. ... } while ( x **<** **10** );



Node.js Underscore Variable

You can also use underscore \_ to get the last result.

**Example:**

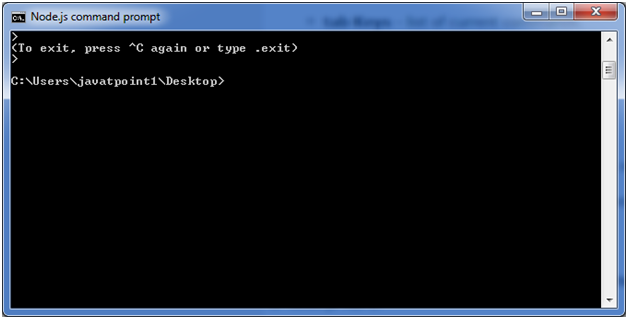


Node.js REPL Commands

|  |  |
| --- | --- |
| Commands | Description |
| ctrl + c | It is used to terminate the current command. |
| ctrl + c twice | It terminates the node repl. |
| ctrl + d | It terminates the node repl. |
| up/down keys | It is used to see command history and modify previous commands. |
| tab keys | It specifies the list of current command. |
| .help | It specifies the list of all commands. |
| .break | It is used to exit from multi-line expressions. |
| .clear | It is used to exit from multi-line expressions. |
| .save filename | It saves current node repl session to a file. |
| .load filename | It is used to load file content in current node repl session. |

Node.js Exit REPL

Use ctrl + c command twice to come out of Node.js REPL.



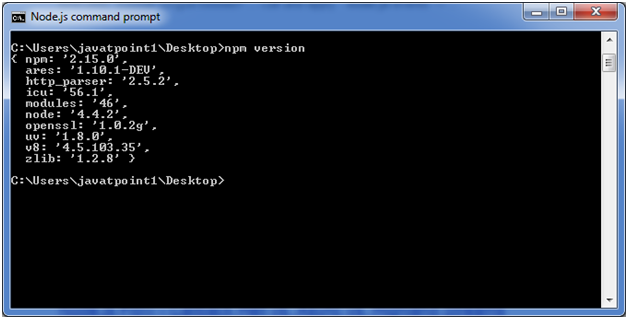
Node.js Package Manager

Node Package Manager provides two main functionalities:

* It provides online repositories for node.js packages/modules which are searchable on search.nodejs.org
* It also provides command line utility to install Node.js packages, do version management and dependency management of Node.js packages.

The npm comes bundled with Node.js installables in versions after that v0.6.3. You can check the version by opening Node.js command prompt and typing the following command:

1. npm  version



Installing Modules using npm

Following is the syntax to install any Node.js module:

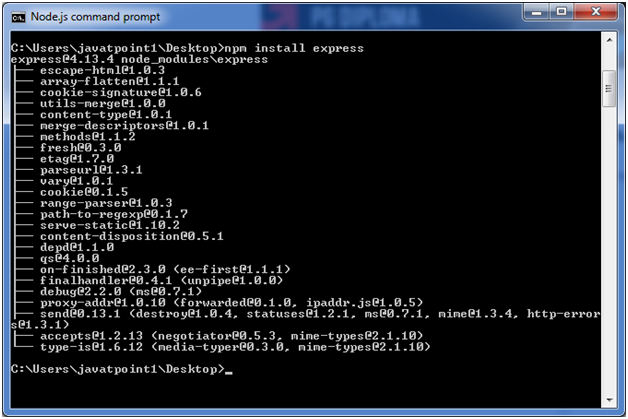
1. npm install **<Module** Name**>**

Let's install a famous Node.js web framework called express:

Open the Node.js command prompt and execute the following command:

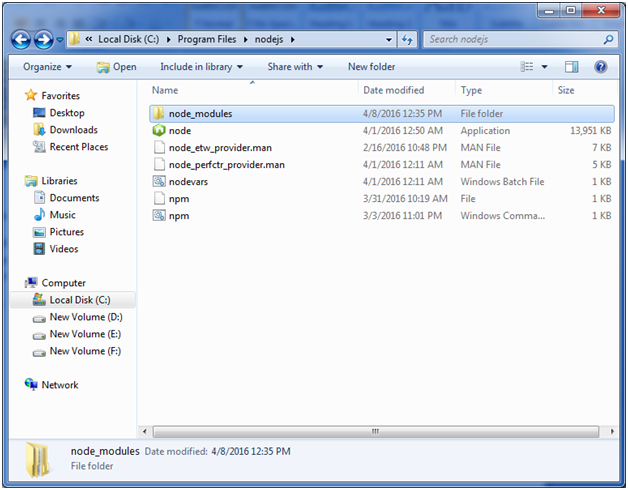
1. npm install express

You can see the result after installing the "express" framework.



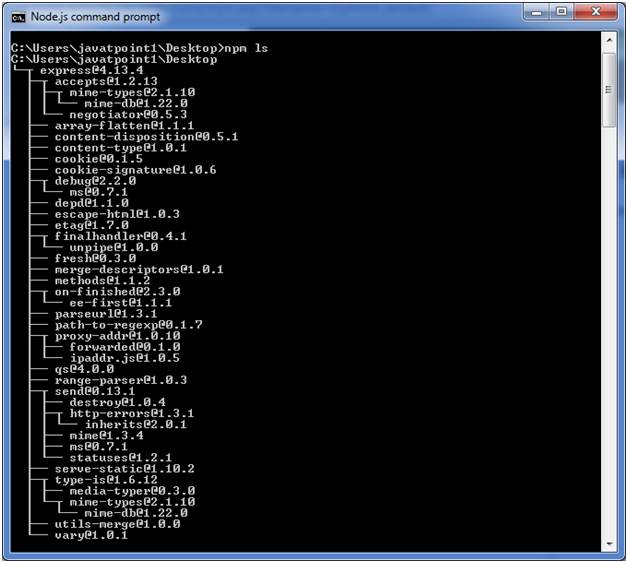
Global vs Local Installation

By default, npm installs dependency in local mode. Here local mode specifies the folder where Node application is present. For example if you installed express module, it created node\_modules directory in the current directory where it installed express module.



You can use npm ls command to list down all the locally installed modules.

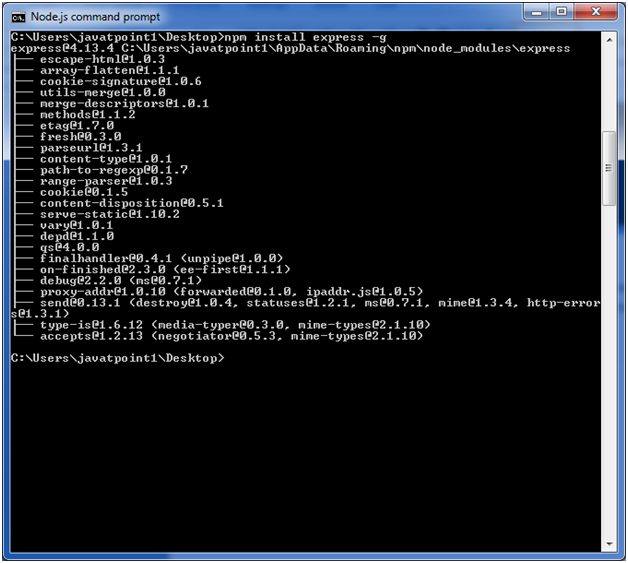
Open the Node.js command prompt and execute "npm ls":



Globally installed packages/dependencies are stored in system directory. Let's install express module using global installation. Although it will also produce the same result but modules will be installed globally.

Open Node.js command prompt and execute the following code:

1. npm install express -g

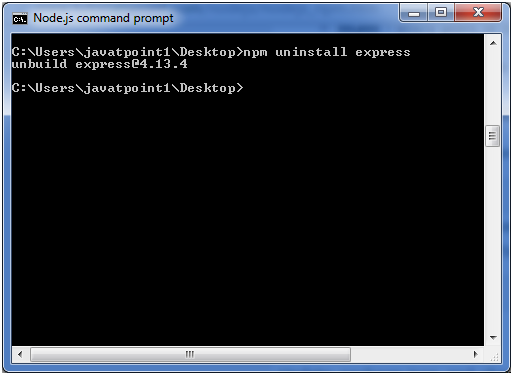


Here first line tells about the module version and its location where it is getting installed.

Uninstalling a Module

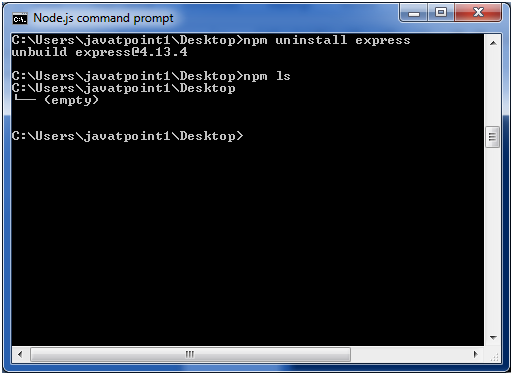
To uninstall a Node.js module, use the following command:

1. npm uninstall express



The Node.js module is uninstalled. You can verify by using the following command:

1. npm ls

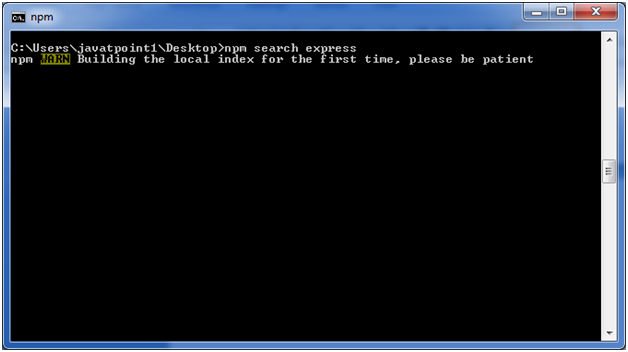
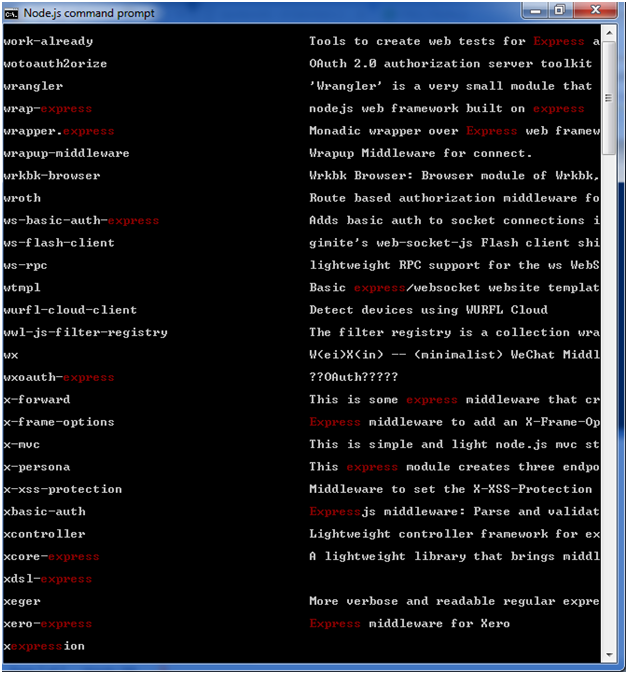


You can see that the module is empty now.

Searching a Module

"npm search express" command is used to search express or module.

1. npm search express

Node.js Command Line Options

There is a wide variety of command line options in Node.js. These options provide multiple ways to execute scripts and other helpful run-time options.

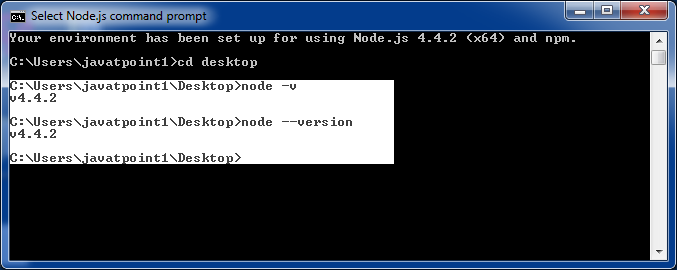
Let's see the list of Node.js command line options:

|  |  |  |
| --- | --- | --- |
| **Index** | **Option** | **Description** |
| 1. | v, --version | It is used to print node's version. |
| 2. | -h, --help | It is used to print node command line options. |
| 3. | -e, --eval "script" | It evaluates the following argument as JavaScript. The modules which are predefined in the REPL can also be used in script. |
| 4. | -p, --print "script" | It is identical to -e but prints the result. |
| 5. | -c, --check | Syntax check the script without executing. |
| 6. | -i, --interactive | It opens the REPL even if stdin does not appear to be a terminal. |
| 7. | -r, --require module | It is used to preload the specified module at startup. It follows require()'s module resolution rules. Module may be either a path to a file, or a node module name. |
| 8. | --no-deprecation | Silence deprecation warnings. |
| 9. | --trace-deprecation | It is used to print stack traces for deprecations. |
| 10. | --throw-deprecation | It throws errors for deprecations. |
| 11. | --no-warnings | It silence all process warnings (including deprecations). |
| 12. | --trace-warnings | It prints stack traces for process warnings (including deprecations). |
| 13. | --trace-sync-io | It prints a stack trace whenever synchronous i/o is detected after the first turn of the event loop. |
| 14. | --zero-fill-buffers | Automatically zero-fills all newly allocated buffer and slowbuffer instances. |
| 15. | --track-heap-objects | It tracks heap object allocations for heap snapshots. |
| 16. | --prof-process | It processes V8 profiler output generated using the v8 option --prof. |
| 17. | --V8-options | It prints V8 command line options. |
| 18. | --tls-cipher-list=list | It specifies an alternative default tls cipher list. (requires node.js to be built with crypto support. (default)) |
| 19. | --enable-fips | It enables fips-compliant crypto at startup. (requires node.js to be built with ./configure --openssl-fips) |
| 20. | --force-fips | It forces fips-compliant crypto on startup. (cannot be disabled from script code.) (same requirements as --enable-fips) |
| 21. | --icu-data-dir=file | It specifies ICU data load path. (Overrides node\_icu\_data) |

Node.js Command Line Options Examples

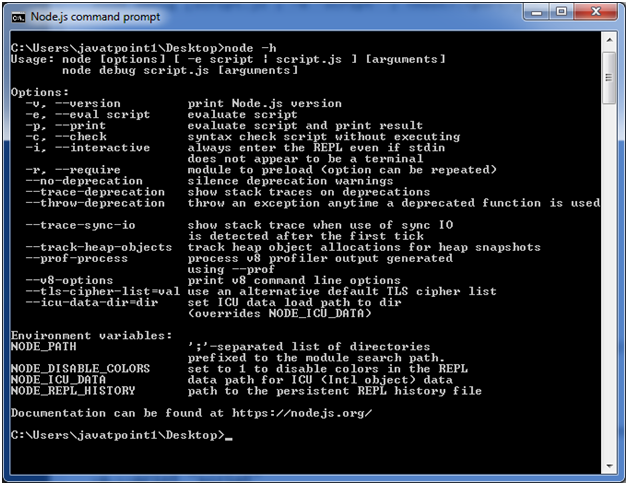
To see the version of the running Node:

Open Node.js command prompt and run command node -v or node --version



For Help:

Use command node ?h or node --help

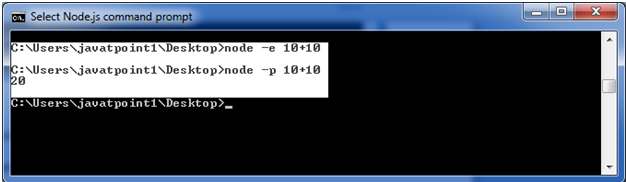


To evaluate an argument (but not print result):

Use command node -e, --eval "script"

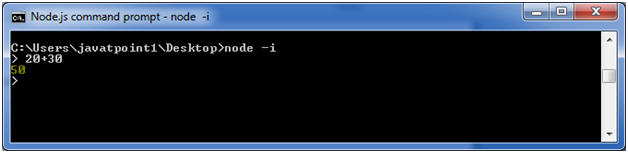
To evaluate an argument and print result also:

Use command node -p "script"



To open REPL even if stdin doesn't appear:

Use command node -i, or node --interactive



Node.js Global Objects

Node.js global objects are global in nature and available in all modules. You don't need to include these objects in your application; rather they can be used directly. These objects are modules, functions, strings and object etc. Some of these objects aren't actually in the global scope but in the module scope.

A list of Node.js global objects are given below:

* \_\_dirname
* \_\_filename
* Console
* Process
* Buffer
* setImmediate(callback[, arg][, ...])
* setInterval(callback, delay[, arg][, ...])
* setTimeout(callback, delay[, arg][, ...])
* clearImmediate(immediateObject)
* clearInterval(intervalObject)
* clearTimeout(timeoutObject)

Node.js \_\_dirname

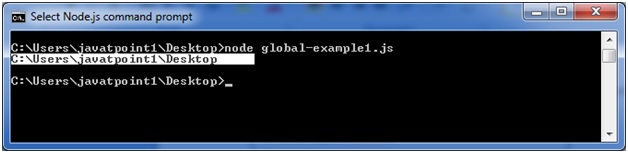
It is a string. It specifies the name of the directory that currently contains the code.

*File: global-example1.js*

1. console.log(\_\_dirname);

Open Node.js command prompt and run the following code:

1. node global-example1.js



Node.js \_\_filename

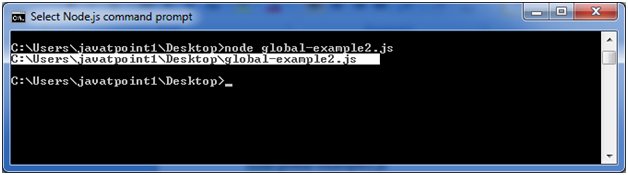
It specifies the filename of the code being executed. This is the resolved absolute path of this code file. The value inside a module is the path to that module file.

*File: global-example2.js*

1. console.log(\_\_filename);

Open Node.js command prompt and run the following code:

1. node global-example2.js



Node.js Console

Click here to get details of Console class. [http://www.javatpoint.com/nodejs-console](https://www.javatpoint.com/nodejs-console)

Node.js Buffer

Click here to get details of Buffer class. [http://www.javatpoint.com/nodejs-buffers](https://www.javatpoint.com/nodejs-buffers)

Node.js Timer Functions

Click here to get details of Timer functions. [http://www.javatpoint.com/nodejs-timer](https://www.javatpoint.com/nodejs-timer)

Node.js OS

Node.js OS provides some basic operating-system related utility functions. Let's see the list generally used functions or methods.

|  |  |  |
| --- | --- | --- |
| **Index** | **Method** | **Description** |
| 1. | os.arch() | This method is used to fetch the operating system CPU architecture. |
| 2. | os.cpus() | This method is used to fetch an array of objects containing information about each cpu/core installed: model, speed (in MHz), and times (an object containing the number of milliseconds the cpu/core spent in: user, nice, sys, idle, and irq). |
| 3. | os.endianness() | This method returns the endianness of the cpu. Its possible values are 'BE' for big endian or 'LE' for little endian. |
| 4. | os.freemem() | This methods returns the amount of free system memory in bytes. |
| 5. | os.homedir() | This method returns the home directory of the current user. |
| 6. | os.hostname() | This method is used to returns the hostname of the operating system. |
| 7. | os.loadavg() | This method returns an array containing the 1, 5, and 15 minute load averages. The load average is a time fraction taken by system activity, calculated by the operating system and expressed as a fractional number. |
| 8. | os.networkinterfaces() | This method returns a list of network interfaces. |
| 9. | os.platform() | This method returns the operating system platform of the running computer i.e.'darwin', 'win32','freebsd', 'linux', 'sunos' etc. |
| 10. | os.release() | This method returns the operating system release. |
| 11. | os.tmpdir() | This method returns the operating system's default directory for temporary files. |
| 12. | os.totalmem() | This method returns the total amount of system memory in bytes. |
| 13. | os.type() | This method returns the operating system name. For example 'linux' on linux, 'darwin' on os x and 'windows\_nt' on windows. |
| 14. | os.uptime() | This method returns the system uptime in seconds. |
| 15. | os.userinfo([options]) | This method returns a subset of the password file entry for the current effective user. |

Node.js OS Example 1

In this example, we are including some basic functions. Create a file named os\_example1.js having the following code:

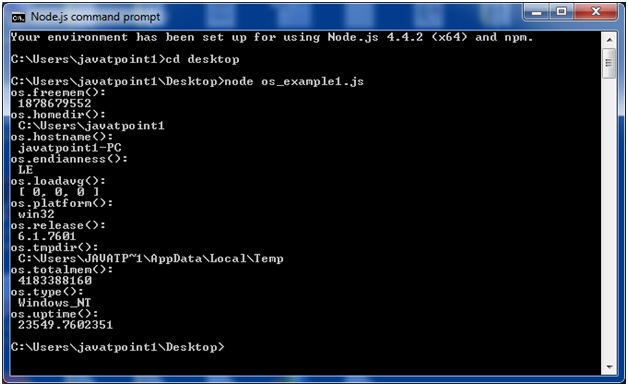
*File: os\_example1.js*

1. const os=require('os');
2. console.log("os.freemem(): \n",os.freemem());
3. console.log("os.homedir(): \n",os.homedir());
4. console.log("os.hostname(): \n",os.hostname());
5. console.log("os.endianness(): \n",os.endianness());
6. console.log("os.loadavg(): \n",os.loadavg());
7. console.log("os.platform(): \n",os.platform());
8. console.log("os.release(): \n",os.release());
9. console.log("os.tmpdir(): \n",os.tmpdir());
10. console.log("os.totalmem(): \n",os.totalmem());
11. console.log("os.type(): \n",os.type());
12. console.log("os.uptime(): \n",os.uptime());

Open Node.js command prompt and run the following code:

Play Video

1. node os\_example1.js



Node.js OS Example 2

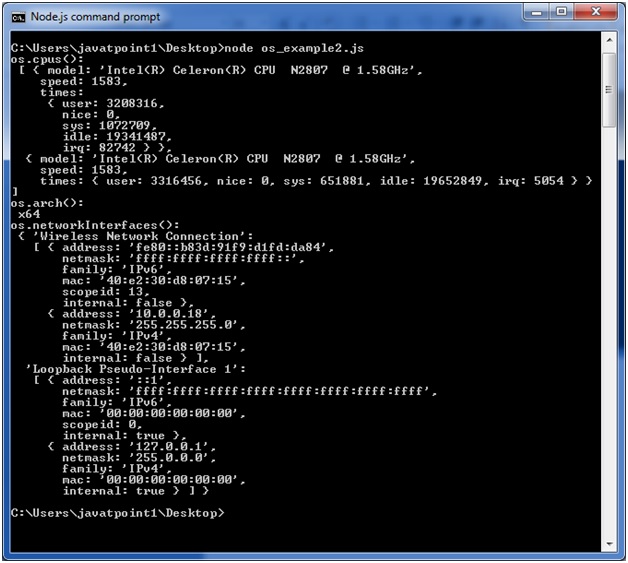
In this example, we are including remaining functions. Create a file named os\_example2.js having the following code:

*File: os\_example2.js*

1. const os=require('os');
2. console.log("os.cpus(): \n",os.cpus());
3. console.log("os.arch(): \n",os.arch());
4. console.log("os.networkInterfaces(): \n",os.networkInterfaces());

Open Node.js command prompt and run the following code:

1. node os\_example2.js



Node.js Timer

Node.js Timer functions are global functions. You don't need to use require() function in order to use timer functions. Let's see the list of timer functions.

**Set timer functions:**

* **setImmediate():** It is used to execute setImmediate.
* **setInterval():** It is used to define a time interval.
* **setTimeout():** ()- It is used to execute a one-time callback after delay milliseconds.

**Clear timer functions:**

* **clearImmediate(immediateObject):** It is used to stop an immediateObject, as created by setImmediate
* **clearInterval(intervalObject):** It is used to stop an intervalObject, as created by setInterval
* **clearTimeout(timeoutObject):** It prevents a timeoutObject, as created by setTimeout

Node.js Timer setInterval() Example

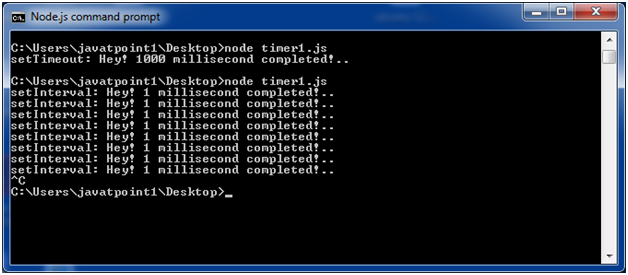
This example will set a time interval of 1000 millisecond and the specified comment will be displayed after every 1000 millisecond until you terminate.

*File: timer1.js*

1. setInterval(function() {
2. console.log("setInterval: Hey! 1 millisecond completed!..");
3. }, 1000);

Open Node.js command prompt and run the following code:

1. node timer1.js

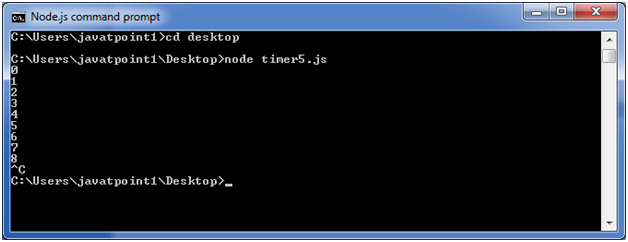


*File: timer5.js*

1. var i =0;
2. console.log(i);
3. setInterval(function(){
4. i++;
5. console.log(i);
6. }, 1000);

Open Node.js command prompt and run the following code:

1. node timer5.js



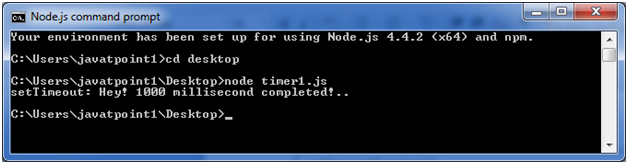
Node.js Timer setTimeout() Example

*File: timer1.js*

1. setTimeout(function() {
2. console.log("setTimeout: Hey! 1000 millisecond completed!..");
3. }, 1000);

Open Node.js command prompt and run the following code:

1. node timer1.js



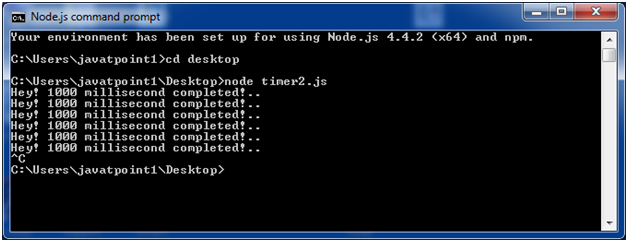
This example shows time out after every 1000 millisecond without setting a time interval. This example uses the recursion property of a function.

*File: timer2.js*

1. var recursive = function () {
2. console.log("Hey! 1000 millisecond completed!..");
3. setTimeout(recursive,1000);
4. }
5. recursive();

Open Node.js command prompt and run the following code:

1. node timer2.js



Node.js setInterval(), setTimeout() and clearTimeout()

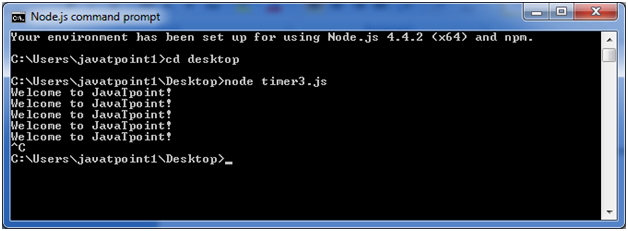
Let's see an example to use clearTimeout() function.

*File: timer3.js*

1. function welcome () {
2. console.log("Welcome to JavaTpoint!");
3. }
4. var id1 = setTimeout(welcome,1000);
5. var id2 = setInterval(welcome,1000);
6. clearTimeout(id1);
7. //clearInterval(id2);

Open Node.js command prompt and run the following code:

1. node timer3.js



You can see that the above example is recursive in nature. It will terminate after one step if you use ClearInterval.

Node.js setInterval(), setTimeout() and clearInterval()

Let's see an example to use clearInterval() function.

*File: timer3.js*

1. function welcome () {
2. console.log("Welcome to JavaTpoint!");
3. }
4. var id1 = setTimeout(welcome,1000);
5. var id2 = setInterval(welcome,1000);
6. //clearTimeout(id1);
7. clearInterval(id2);

Open Node.js command prompt and run the following code:

1. node timer3.js



Node.js Errors

The Node.js applications generally face four types of errors:

* **Standard JavaScript errors** i.e. <EvalError>, <SyntaxError>, <RangeError>, <ReferenceError>, <TypeError>, <URIError> etc.
* **System errors**
* **User-specified errors**
* **Assertion errors**

Node.js Errors Example 1

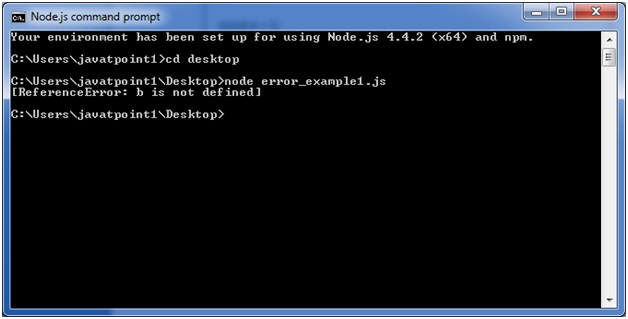
Let's take an example to deploy standard JavaScript error - ReferenceError.

*File: error\_example1.js*

1. // Throws with a ReferenceError because b is undefined
2. **try** {
3. **const** a = 1;
4. **const** c = a + b;
5. } **catch** (err) {
6. console.log(err);
7. }

Open Node.js command prompt and run the following code:

1. node error\_example1.js



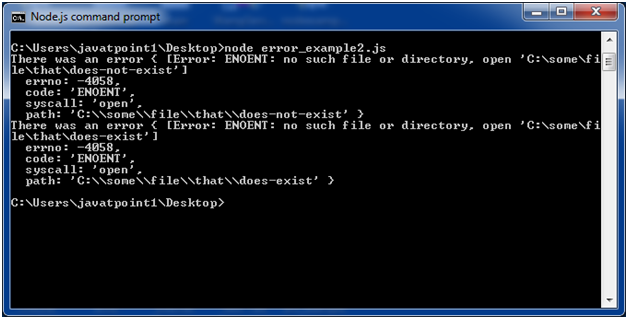
Node.js Errors Example 2

*File: timer2.js*

1. **const** fs = require('fs');
2. function nodeStyleCallback(err, data) {
3. **if** (err) {
4. console.error('There was an error', err);
5. **return**;
6. }
7. console.log(data);
8. }
9. fs.readFile('/some/file/that/does-not-exist', nodeStyleCallback);
10. fs.readFile('/some/file/that/does-exist', nodeStyleCallback);

Open Node.js command prompt and run the following code:

1. node error\_example2.js



Node.js DNS

The Node.js DNS module contains methods to get information of given hostname. Let's see the list of commonly used DNS functions:

* dns.getServers()
* dns.setServers(servers)
* dns.lookup(hostname[, options], callback)
* dns.lookupService(address, port, callback)
* dns.resolve(hostname[, rrtype], callback)
* dns.resolve4(hostname, callback)
* dns.resolve6(hostname, callback)
* dns.resolveCname(hostname, callback)
* dns.resolveMx(hostname, callback)
* dns.resolveNs(hostname, callback)
* dns.resolveSoa(hostname, callback)
* dns.resolveSrv(hostname, callback)
* dns.resolvePtr(hostname, callback)
* dns.resolveTxt(hostname, callback)
* dns.reverse(ip, callback)

Node.js DNS Example 1

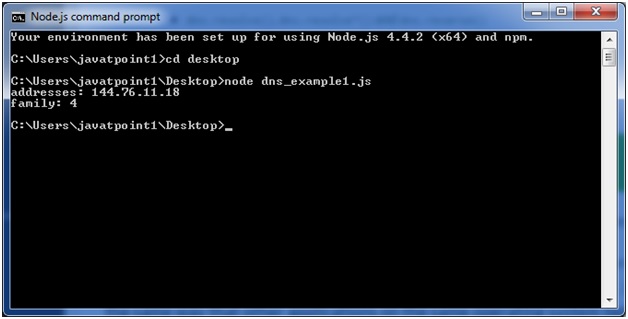
Let's see the example of dns.lookup() function.

*File: dns\_example1.js*

1. const dns = require('dns');
2. dns.lookup('www.javatpoint.com', (err, addresses, family) =**>** {
3. console.log('addresses:', addresses);
4. console.log('family:',family);
5. });

Open Node.js command prompt and run the following code:

1. node dns\_example1.js



Node.js DNS Example 2

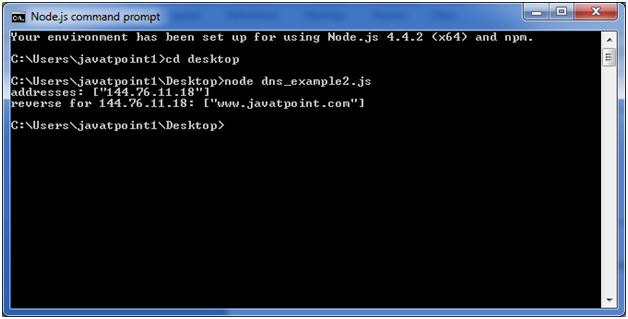
Let's see the example of resolve4() and reverse() functions.

*File: dns\_example2.js*

1. const dns = require('dns');
2. dns.resolve4('www.javatpoint.com', (err, addresses) =**>** {
3. if (err) throw err;
4. console.log(`addresses: ${JSON.stringify(addresses)}`);
5. addresses.forEach((a) =**>** {
6. dns.reverse(a, (err, hostnames) =**>** {
7. if (err) {
8. throw err;
9. }
10. console.log(`reverse for ${a}: ${JSON.stringify(hostnames)}`);
11. });
12. });
13. });

Open Node.js command prompt and run the following code:

1. node dns\_example2.js



Node.js DNS Example 3

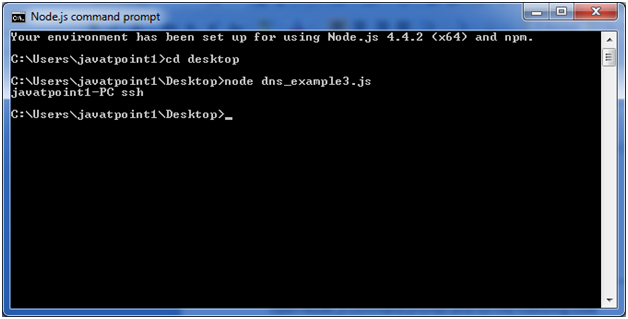
Let's take an example to print the localhost name using lookupService() function.

*File: dns\_example3.js*

1. const dns = require('dns');
2. dns.lookupService('127.0.0.1', 22, (err, hostname, service) =**>** {
3. console.log(hostname, service);
4. // Prints: localhost
5. });

Open Node.js command prompt and run the following code:

1. node dns\_example3.js



# Node.js Net

Node.js provides the ability to perform socket programming. We can create chat application or communicate client and server applications using socket programming in Node.js. The Node.js net module contains functions for creating both servers and clients.

## Node.js Net Example

In this example, we are using two command prompts:

* Node.js command prompt for server.
* Window's default command prompt for client.

**server:**

*File: net\_server.js*

1. **const** net = require('net');
2. var server = net.createServer((socket) => {
3. socket.end('goodbye\n');
4. }).on('error', (err) => {
5. // handle errors here
6. **throw** err;
7. });
8. // grab a random port.
9. server.listen(() => {
10. address = server.address();
11. console.log('opened server on %j', address);
12. });

Open Node.js command prompt and run the following code:

1. node net\_server.js

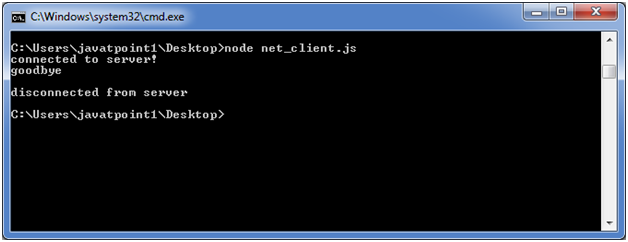
**client:**

*File: net\_client.js*

1. **const** net = require('net');
2. **const** client = net.connect({port: 50302}, () => {//use same port of server
3. console.log('connected to server!');
4. client.write('world!\r\n');
5. });
6. client.on('data', (data) => {
7. console.log(data.toString());
8. client.end();
9. });
10. client.on('end', () => {
11. console.log('disconnected from server');
12. });

Open Node.js command prompt and run the following code:

1. node net\_client.js



#### **Note: You must match the port. The client and server should have similar port for successful connection.**

Node.js Crypto

The Node.js Crypto module supports cryptography. It provides cryptographic functionality that includes a set of wrappers for open SSL's hash HMAC, cipher, decipher, sign and verify functions.

What is Hash

A hash is a fixed-length string of bits i.e. procedurally and deterministically generated from some arbitrary block of source data.

What is HMAC

HMAC stands for Hash-based Message Authentication Code. It is a process for applying a hash algorithm to both data and a secret key that results in a single final hash.

Encryption Example using Hash and HMAC

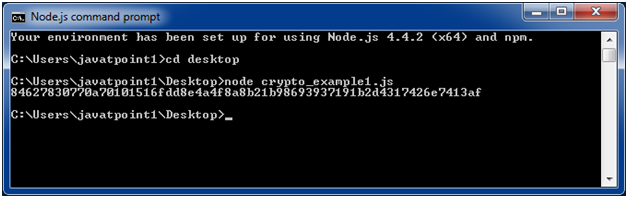
*File: crypto\_example1.js*

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. **const** crypto = require('crypto');
2. **const** secret = 'abcdefg';
3. **const** hash = crypto.createHmac('sha256', secret)
4. .update('Welcome to JavaTpoint')
5. .digest('hex');
6. console.log(hash);

Open Node.js command prompt and run the following code:

1. node crypto\_example1.js



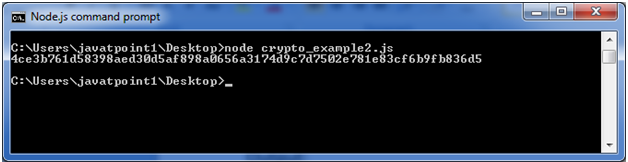
Encryption example using Cipher

*File: crypto\_example2.js*

1. **const** crypto = require('crypto');
2. **const** cipher = crypto.createCipher('aes192', 'a password');
3. var encrypted = cipher.update('Hello JavaTpoint', 'utf8', 'hex');
4. encrypted += cipher.**final**('hex');
5. console.log(encrypted);

Open Node.js command prompt and run the following code:

1. node crypto\_example2.js



Decryption example using Decipher

*File: crypto\_example3.js*

1. **const** crypto = require('crypto');
2. **const** decipher = crypto.createDecipher('aes192', 'a password');
3. var encrypted = '4ce3b761d58398aed30d5af898a0656a3174d9c7d7502e781e83cf6b9fb836d5';
4. var decrypted = decipher.update(encrypted, 'hex', 'utf8');
5. decrypted += decipher.**final**('utf8');
6. console.log(decrypted);

Open Node.js command prompt and run the following code:

1. node crypto\_example3.js



Node.js TLS/SSL

What is TLS/SSL

TLS stands for Transport Layer Security. It is the successor to Secure Sockets Layer (SSL). TLS along with SSL is used for cryptographic protocols to secure communication over the web.

TLS uses public-key cryptography to encrypt messages. It encrypts communication generally on the TCP layer.

What is public-key cryptography

In public-key cryptography, each client and each server has two keys: public key and private key. Public key is shared with everyone and private key is secured. To encrypt a message, a computer requires its private key and the recipient?s public key. On the other hand, to decrypt the message, the recipient requires its own

You have to use **require('tls')** to access this module.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

**Syntax:**

1. var tls = require('tls');

The tls module uses OpenSSL to attain Transport Layer Security and Secure Socket Layer. TLS/SSL is a public/private key infrastructure. Each client and each server must have a private key.

A private key can be created like this:

1. openssl genrsa -out ryans-key.pem 1024

All severs and some clients need to have a certificate. Certificates are public keys signed by a Certificate Authority or self-signed. To get certificate, you have to create a "Certificate Signing Request" (CSR) file.

A certificate can be created like this:

1. openssl req -new -key ryans-key.pem -out ryans-csr.pem

To create a self-signed certificate with the CSR:

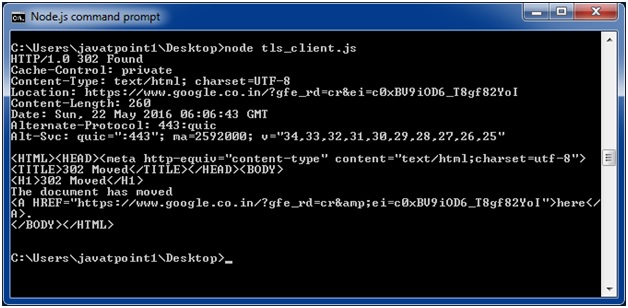
1. openssl x509 -req -in ryans-csr.pem -signkey ryans-key.pem -out ryans-cert.pem

Node.js TLS client example

*File: tls\_client.js*

1. tls = require('tls');
2. function connected(stream) {
3. **if** (stream) {
4. // socket connected
5. stream.write("GET / HTTP/1.0\n\rHost: encrypted.google.com:443\n\r\n\r");
6. } **else** {
7. console.log("Connection failed");
8. }
9. }
10. // needed to keep socket variable in scope
11. var dummy = **this**;
12. // try to connect to the server
13. dummy.socket = tls.connect(443, 'encrypted.google.com', function() {
14. // callback called only after successful socket connection
15. dummy.connected = **true**;
16. **if** (dummy.socket.authorized) {
17. // authorization successful
18. dummy.socket.setEncoding('utf-8');
19. connected(dummy.socket);
20. } **else** {
21. // authorization failed
22. console.log(dummy.socket.authorizationError);
23. connected(**null**);
24. }
25. });
26. dummy.socket.addListener('data', function(data) {
27. // received data
28. console.log(data);
29. });
30. dummy.socket.addListener('error', function(error) {
31. **if** (!dummy.connected) {
32. // socket was not connected, notify callback
33. connected(**null**);
34. }
35. console.log("FAIL");
36. console.log(error);
37. });
38. dummy.socket.addListener('close', function() {
39. // do something
40. });

**Output:**



Node.js Debugger

Node.js provides a simple TCP based protocol and built-in debugging client. For debugging your JavaScript file, you can use debug argument followed by the js file name you want to debug.

**Syntax:**

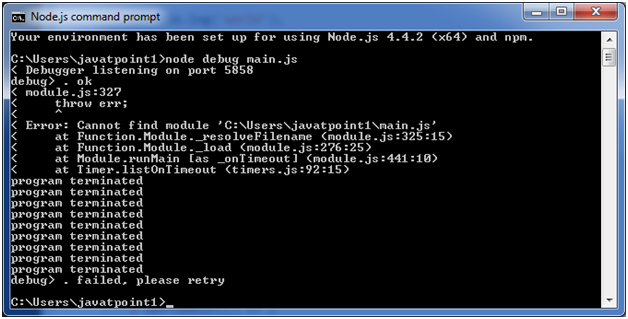
1. node debug [script.js | -e "script" | <host>:<port>]

**Example:**

1. node debug main.js

**If you make any error:**

If you make any error in your js file source code or provide a wrong path on the Node.js command prompt then you will get the following result.



In the above example, the path is not specified although the main.js file is located on desktop.

Node.js Process

Node.js provides the facility to get process information such as process id, architecture, platform, version, release, uptime, upu usage etc. It can also be used to kill process, set uid, set groups, unmask etc.

The process is a global object, an instance of EventEmitter, can be accessed from anywhere.

Node.js Process Properties

A list of commonly used Node.js process properties are given below.

|  |  |
| --- | --- |
| **Property** | **Description** |
| arch | returns process architecture: 'arm', 'ia32', or 'x64' |
| args | returns commands line arguments as an array |
| env | returns user environment |
| pid | returns process id of the process |
| platform | returns platform of the process: 'darwin', 'freebsd', 'linux', 'sunos' or 'win32' |
| release | returns the metadata for the current node release |
| version | returns the node version |
| versions | returns the node version and its dependencies |

Node.js Process Properties Example

Let's see the simple process **example to print architecture, pid, platform and version** of the process.

*File: process\_example1.js*

1. console.log(`Process Architecture: ${process.arch}`);
2. console.log(`Process PID: ${process.pid}`);
3. console.log(`Process Platform: ${process.platform}`);
4. console.log(`Process Version: ${process.version}`);

Open Node.js command prompt and run the following code:

1. node process\_example1.js



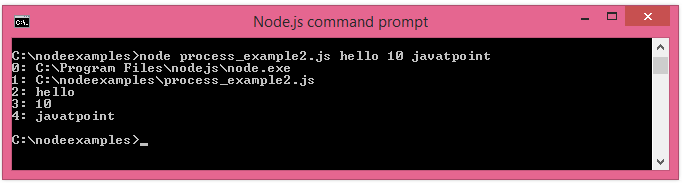
Let's see another process **example to print command line arguments**. Here node is considered as the first argument, filename is considered as the second argument and actual command line arguments are considered as third, fourth, fifth and so on.

*File: process\_example2.js*

1. process.argv.forEach((value, index, array) => {
2. console.log(`${index}: ${value}`);
3. });

Open Node.js command prompt and run the following code:

1. node process\_example2.js



Node.js Process Functions

A list of commonly used Node.js process functions are given below.

|  |  |
| --- | --- |
| **Function** | **Description** |
| cwd() | returns path of current working directory |
| hrtime() | returns the current high-resolution real time in a [seconds, nanoseconds] array |
| memoryUsage() | returns an object having information of memory usage. |
| process.kill(pid[, signal]) | is used to kill the given pid. |
| uptime() | returns the Node.js process uptime in seconds. |

Node.js Process Functions Example

Let's see the process **example to print current working directory and uptime** of the process.

*File: process\_example3.js*

1. console.log(`Current directory: ${process.cwd()}`);
2. console.log(`Uptime: ${process.uptime()}`);

Open Node.js command prompt and run the following code:

1. node process\_example3.js



Node.js Child Process

The Node.js child process module provides the ability to spawn child processes in a similar manner to popen(3).

There are three major way to create child process:

* **child\_process.exec() method:** This method runs a command in a console and buffers the output.
* **child\_process.spawn() method:** This method launches a new process with a given command.
* **child\_process.fork() method:** This method is a special case of spawn() method to create child processes.

Node.js child\_process.exec() method

The child\_process.exec() method runs a command in a console and buffers the output.

**Syntax:**

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. child\_process.exec(command[, options], callback)

**Parameters:**

1) command: It specifies the command to run, with space-separated arguments.

2) options: It may contain one or more of the following options:

* **cwd:** It specifies the current working directory of the child process.
* **env:** It specifies environment key-value pairs.
* **encoding:** String (Default: 'utf8')
* **shell:** It specifies string Shell to execute the command with (Default: '/bin/sh' on UNIX, 'cmd.exe' on Windows, The shell should understand the -c switch on UNIX or /s /c on Windows. On Windows, command line parsing should be compatible with cmd.exe.)
* **timeout:** Number (Default: 0)
* **maxBuffer:** Number (Default: 200\*1024)
* **killSignal:** String (Default: 'SIGTERM')
* **uid Number:** Sets the user identity of the process.
* **gid Number:** Sets the group identity of the process.

**callback:** The callback function specifies three arguments error, stdout and stderr which is called with the following output when process terminates.

Node.js child\_process.exec() example 1

Let's see the simple process **example to print architecture, pid, platform and version** of the process.

*File: child\_process\_example1.js*

1. **const** exec = require('child\_process').exec;
2. exec('my.bat', (err, stdout, stderr) => {
3. **if** (err) {
4. console.error(err);
5. **return**;
6. }
7. console.log(stdout);
8. });

Create a batch file named my.bat having the following code:

*File: my.bat*

1. dir
2. mkdir child

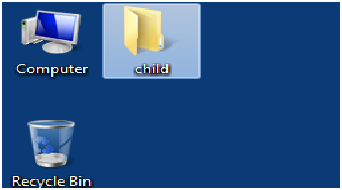
Open Node.js command prompt and run the following code:

1. node child\_process\_example1.js

It will execute two commands *dir* and *mkdir child*. The dir command will display list of current directory and mkdir command will create a new directory. For linux, you can you ls command to display the current directory list.



It will create a new directory also.



Node.js child\_process.exec() example 2

Create two js files named support.js and master.js, having the following code:

*File: support.js*

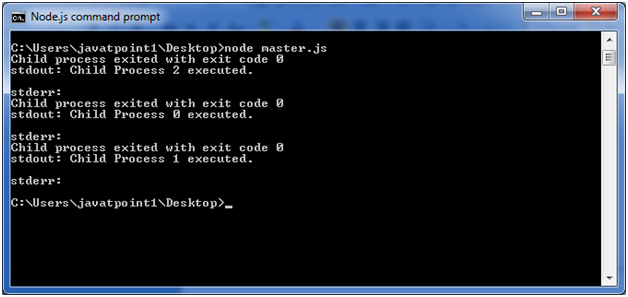
1. console.log("Child Process " + process.argv[2] + " executed." );

*File: master.js*

1. **const** fs = require('fs');
2. **const** child\_process = require('child\_process');
3. **for**(var i=0; i<3; i++) {
4. var workerProcess = child\_process.exec('node support.js '+i,
5. function (error, stdout, stderr) {
6. **if** (error) {
7. console.log(error.stack);
8. console.log('Error code: '+error.code);
9. console.log('Signal received: '+error.signal);
10. }
11. console.log('stdout: ' + stdout);
12. console.log('stderr: ' + stderr);
13. });
14. workerProcess.on('exit', function (code) {
15. console.log('Child process exited with exit code '+code);
16. });
17. }

Open Node.js command prompt and run the following code:

1. node master.js



Node.js child\_process.spawn() method

The child\_process.spawn() method launches a new process with a given command. This method returns streams (stdout & stderr) and it is generally used when the process returns large amount of data.

**Syntax:**

1. child\_process.spawn(command[, args][, options])

**Parameters:**

1) command: It specifies the command to run.

2) args: It specifies an array List of string arguments.

3) options: It may contain one or more of the following options:

* **cwd:** It specifies the current working directory of the child process.
* **env:** It specifies environment key-value pairs.
* **stdio:** Array|String Child's stdio configuration
* **customFds:** Array Deprecated File descriptors for the child to use for stdio
* **detached Boolean :** The child will be a process group leader
* **uid Number:** Sets the user identity of the process.
* **gid Number:** Sets the group identity of the process.

Node.js child\_process.spawn() example

Create two js files named support.js and master.js, having the following code:

*File: support.js*

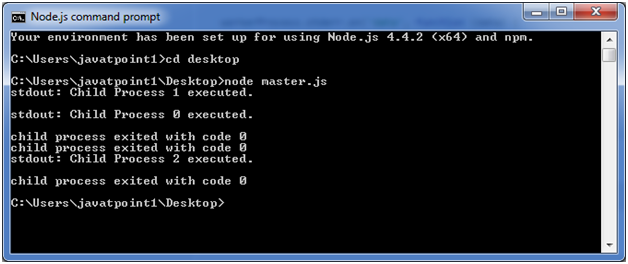
1. console.log("Child Process " + process.argv[2] + " executed." );

*File: master.js*

1. **const** fs = require('fs');
2. **const** child\_process = require('child\_process');
3. **for**(var i=0; i<3; i++) {
4. var workerProcess = child\_process.spawn('node', ['support.js', i]);
5. workerProcess.stdout.on('data', function (data) {
6. console.log('stdout: ' + data);
7. });
8. workerProcess.stderr.on('data', function (data) {
9. console.log('stderr: ' + data);
10. });
11. workerProcess.on('close', function (code) {
12. console.log('child process exited with code ' + code);
13. });
14. }

Open Node.js command prompt and run the following code:

1. node master.js



Node.js child\_process.fork() method

The child\_process.fork method is a special case of the spawn() to create Node processes. This method returns object with a built-in communication channel in addition to having all the methods in a normal ChildProcess instance.

**Syntax:**

1. child\_process.fork(modulePath[, args][, options])

**Parameters:**

1) modulePath: This is a string specifies the module to run in the child.

2) args: It specifies an array List of string arguments.

3) options: It may contain one or more of the following options:

* **cwd:** It specifies the current working directory of the child process.
* **env:** It specifies environment key-value pairs.
* **execPath:** This is a string Executable used to create the child process.
* **execArgv:** It specifies Array List of string arguments passed to the executable (Default: process.execArgv).
* **silent:** It specifies Boolean If true, stdin, stdout, and stderr of the child will be piped to the parent, otherwise they will be inherited from the parent, see the "pipe" and "inherit" options for spawn()'s stdio for more details (default is false).
* **uid Number:** Sets the user identity of the process.
* **gid Number:** Sets the group identity of the process.

Node.js child\_process.fork() example

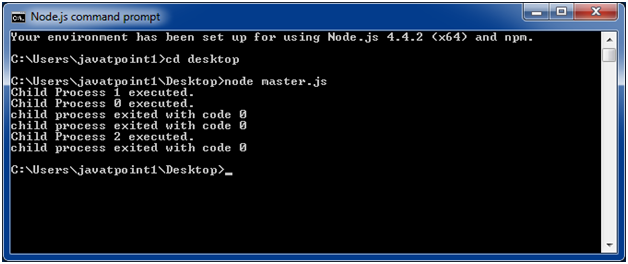
Create two js files named support.js and master.js, having the following code:

*File: support.js*

1. **const** fs = require('fs');
2. **const** child\_process = require('child\_process');
3. **for**(var i=0; i<3; i++) {
4. var worker\_process = child\_process.fork("support.js", [i]);
5. worker\_process.on('close', function (code) {
6. console.log('child process exited with code ' + code);
7. });
8. }

Open Node.js command prompt and run the following code:

1. node master.js



Node.js Buffers

Node.js provides Buffer class to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap. Buffer class is used because pure JavaScript is not nice to binary data. So, when dealing with TCP streams or the file system, it's necessary to handle octet streams.

Buffer class is a global class. It can be accessed in application without importing buffer module.

Node.js Creating Buffers

There are many ways to construct a Node buffer. Following are the three mostly used methods:

1. **Create an uninitiated buffer:** Following is the syntax of creating an uninitiated buffer of 10 octets:
   1. var buf = new Buffer(10);
2. **Create a buffer from array:**Following is the syntax to create a Buffer from a given array:
   1. var buf = new Buffer([10, 20, 30, 40, 50]);
3. **Create a buffer from string:**Following is the syntax to create a Buffer from a given string and optionally encoding type:
   1. var buf = new Buffer("Simply Easy Learning", "utf-8");

Node.js Writing to buffers

Following is the method to write into a Node buffer:

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

**Syntax:**

1. buf.write(string[, offset][, length][, encoding])

**Parameter explanation:**

**string:** It specifies the string data to be written to buffer.

**offset:** It specifies the index of the buffer to start writing at. Its default value is 0.

**length:** It specifies the number of bytes to write. Defaults to buffer.length

**encoding:** Encoding to use. 'utf8' is the default encoding.

**Return values from writing buffers:**

This method is used to return number of octets written. In the case of space shortage for buffer to fit the entire string, it will write a part of the string.

**Let's take an example:**

Create a JavaScript file named "main.js" having the following code:

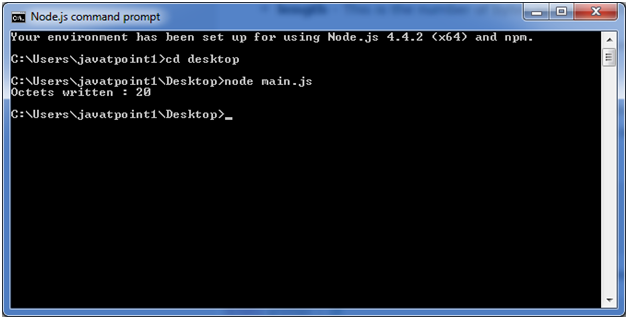
*File: main.js*

1. buf = new Buffer(256);
2. len = buf.write("Simply Easy Learning");
3. console.log("Octets written : "+  len);

Open the Node.js command prompt and execute the following code:

1. node main.js

**Output:**



Node.js Reading from buffers

Following is the method to read data from a Node buffer.

**Syntax:**

1. buf.toString([encoding][, start][, end])

**Parameter explanation:**

**encoding:** It specifies encoding to use. 'utf8' is the default encoding

**start:** It specifies beginning index to start reading, defaults to 0.

**end:**It specifies end index to end reading, defaults is complete buffer.

**Return values reading from buffers:**

This method decodes and returns a string from buffer data encoded using the specified character set encoding.

Let's take an example:

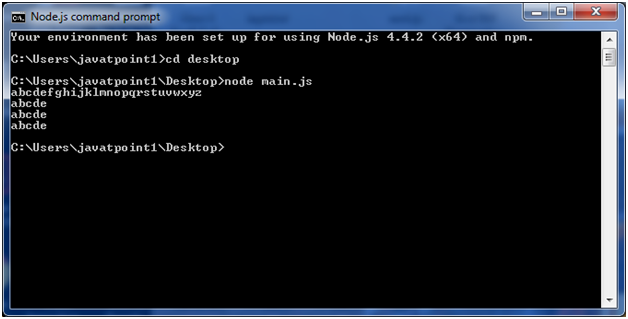
*File: main.js*

1. buf = new Buffer(26);
2. for (var i = 0 ; i **<** **26** ; i++) {
3. buf[i] = i + 97;
4. }
5. console.log( buf.toString('ascii'));       // outputs: abcdefghijklmnopqrstuvwxyz
6. console.log( buf.toString('ascii',0,5));   // outputs: abcde
7. console.log( buf.toString('utf8',0,5));    // outputs: abcde
8. console.log( buf.toString(undefined,0,5)); // encoding defaults to 'utf8', outputs abcde

Open Node.js command prompt and execute the following code:

1. node main.js

**Output:**



Node.js Streams

Streams are the objects that facilitate you to read data from a source and write data to a destination. There are four types of streams in Node.js:

* **Readable:**This stream is used for read operations.
* **Writable:**This stream is used for write operations.
* **Duplex:**This stream can be used for both read and write operations.
* **Transform:**It is type of duplex stream where the output is computed according to input.

Each type of stream is an Event emitter instance and throws several events at different times. Following are some commonly used events:

* **Data:**This event is fired when there is data available to read.
* **End:**This event is fired when there is no more data available to read.
* **Error:**This event is fired when there is any error receiving or writing data.
* **Finish:**This event is fired when all data has been flushed to underlying system.

Node.js Reading from stream

Create a text file named input.txt having the following content:

1. Javatpoint is a one of the best online tutorial website to learn different technologies in a very easy and efficient manner.

Create a JavaScript file named main.js having the following code:

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

*File: main.js*

1. var fs = require("fs");
2. var data = '';
3. // Create a readable stream
4. var readerStream = fs.createReadStream('input.txt');
5. // Set the encoding to be utf8.
6. readerStream.setEncoding('UTF8');
7. // Handle stream events --**>** data, end, and error
8. readerStream.on('data', function(chunk) {
9. data += chunk;
10. });
11. readerStream.on('end',function(){
12. console.log(data);
13. });
14. readerStream.on('error', function(err){
15. console.log(err.stack);
16. });
17. console.log("Program Ended");

Now, open the Node.js command prompt and run the main.js

1. node main.js

**Output:**



Node.js Writing to stream

Create a JavaScript file named main.js having the following code:

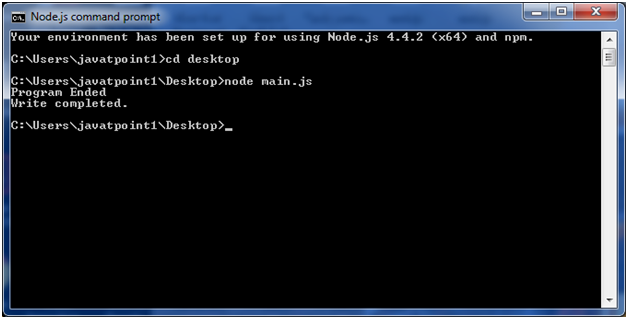
*File: main.js*

1. var fs = require("fs");
2. var data = 'A Solution of all Technology';
3. // Create a writable stream
4. var writerStream = fs.createWriteStream('output.txt');
5. // Write the data to stream with encoding to be utf8
6. writerStream.write(data,'UTF8');
7. // Mark the end of file
8. writerStream.end();
9. // Handle stream events --**>** finish, and error
10. writerStream.on('finish', function() {
11. console.log("Write completed.");
12. });
13. writerStream.on('error', function(err){
14. console.log(err.stack);
15. });
16. console.log("Program Ended");

Now open the Node.js command prompt and run the main.js

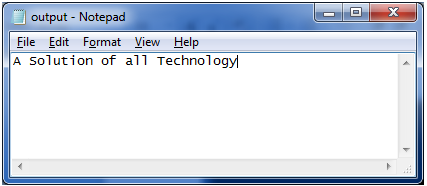
1. node main.js

You will see the following result:



Now, you can see that a text file named "output.txt" is created where you had saved "input.txt" and "main.js" file. In my case, it is on desktop.

Open the "output.txt" and you will see the following content.



Node.js Piping Streams

Piping is a mechanism where output of one stream is used as input to another stream. There is no limit on piping operation.

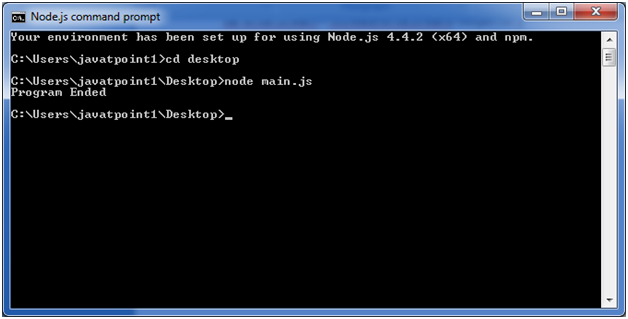
Let's take a piping example for reading from one file and writing it to another file.

*File: main.js*

1. var fs = require("fs");
2. // Create a readable stream
3. var readerStream = fs.createReadStream('input.txt');
4. // Create a writable stream
5. var writerStream = fs.createWriteStream('output.txt');
6. // Pipe the read and write operations
7. // read input.txt and write data to output.txt
8. readerStream.pipe(writerStream);
9. console.log("Program Ended");

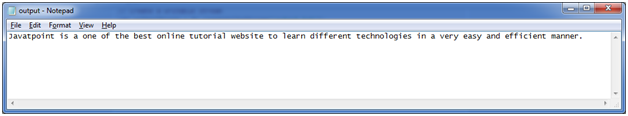
Open the Node.js and run the mian.js

1. node main.js



Now, you can see that a text file named "output.txt" is created where you had saved ?main.js? file. In my case, it is on desktop.

Open the "output.txt" and you will see the following content.



Node.js Chaining Streams

Chaining stream is a mechanism of creating a chain of multiple stream operations by connecting output of one stream to another stream. It is generally used with piping operation.

Let's take an example of piping and chaining to compress a file and then decompress the same file.

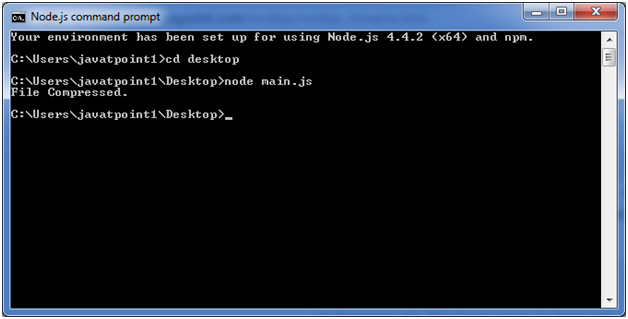
*File: main.js*

1. var fs = require("fs");
2. var zlib = require('zlib');
3. // Compress the file input.txt to input.txt.gz
4. fs.createReadStream('input.txt')
5. .pipe(zlib.createGzip())
6. .pipe(fs.createWriteStream('input.txt.gz'));
7. console.log("File Compressed.");

Open the Node.js command prompt and run main.js

1. node main.js

You will get the following result:



Now you will see that file "input.txt" is compressed and a new file is created named "input.txt.gz" in the current file.

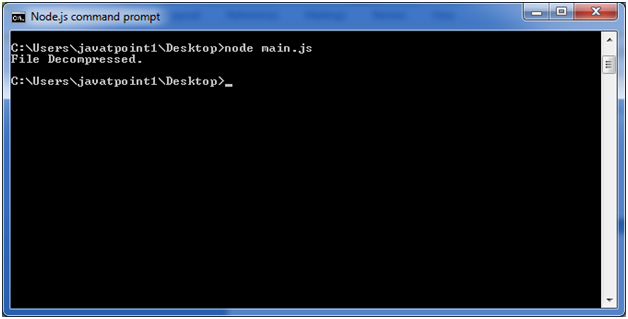
**To Decompress the same file:** put the following code in the js file "main.js"

*File: main.js*

1. var fs = require("fs");
2. var zlib = require('zlib');
3. // Decompress the file input.txt.gz to input.txt
4. fs.createReadStream('input.txt.gz')
5. .pipe(zlib.createGunzip())
6. .pipe(fs.createWriteStream('input.txt'));
7. console.log("File Decompressed.");

Open the Node.js command prompt and run main.js

1. node main.js



Node.js File System (FS)

In Node.js, file I/O is provided by simple wrappers around standard POSIX functions. Node File System (fs) module can be imported using following syntax:

**Syntax:**

1. var fs = require("fs")

Node.js FS Reading File

Every method in fs module has synchronous and asynchronous forms.

Asynchronous methods take a last parameter as completion function callback. Asynchronous method is preferred over synchronous method because it never blocks the program execution where as the synchronous method blocks.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

**Let's take an example:**

Create a text file named "input.txt" having the following content.

*File: input.txt*

1. Javatpoint is a one of the best online tutorial website to learn different technologies
2. in a very easy and efficient manner.

Let's take an example to create a JavaScript file named "main.js" having the following code:

*File: main.js*

1. var fs = require("fs");
2. // Asynchronous read
3. fs.readFile('input.txt', function (err, data) {
4. if (err) {
5. return console.error(err);
6. }
7. console.log("Asynchronous read: " + data.toString());
8. });
9. // Synchronous read
10. var data = fs.readFileSync('input.txt');
11. console.log("Synchronous read: " + data.toString());
12. console.log("Program Ended");

Open Node.js command prompt and run the main.js:

1. node main.js

Node.js Open a file

**Syntax:**

Following is the syntax of the method to open a file in asynchronous mode:

1. fs.open(path, flags[, mode], callback)

**Parameter explanation:**

Following is the description of parameters used in the above syntax:

**path:** This is a string having file name including path.

**flags:** Flag specifies the behavior of the file to be opened. All possible values have been mentioned below.

**mode:** This sets the file mode (permission and sticky bits), but only if the file was created. It defaults to 0666, readable and writeable.

**callback:**This is the callback function which gets two arguments (err, fd).

Node.js Flags for Read/Write

Following is a list of flags for read/write operation:

|  |  |
| --- | --- |
| **Flag** | **Description** |
| r | open file for reading. an exception occurs if the file does not exist. |
| r+ | open file for reading and writing. an exception occurs if the file does not exist. |
| rs | open file for reading in synchronous mode. |
| rs+ | open file for reading and writing, telling the os to open it synchronously. see notes for 'rs' about using this with caution. |
| w | open file for writing. the file is created (if it does not exist) or truncated (if it exists). |
| wx | like 'w' but fails if path exists. |
| w+ | open file for reading and writing. the file is created (if it does not exist) or truncated (if it exists). |
| wx+ | like 'w+' but fails if path exists. |
| a | open file for appending. the file is created if it does not exist. |
| ax | like 'a' but fails if path exists. |
| a+ | open file for reading and appending. the file is created if it does not exist. |
| ax+ | open file for reading and appending. the file is created if it does not exist. |

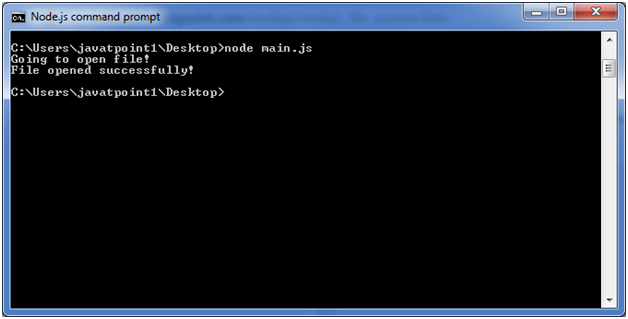
Create a JavaScript file named "main.js" having the following code to open a file input.txt for reading and writing.

*File: main.js*

1. var fs = require("fs");
2. // Asynchronous - Opening File
3. console.log("Going to open file!");
4. fs.open('input.txt', 'r+', function(err, fd) {
5. if (err) {
6. return console.error(err);
7. }
8. console.log("File opened successfully!");
9. });

Open Node.js command prompt and run the main.js:

1. node main.js



Node.js File Information Method

**Syntax:**

Following is syntax of the method to get file information.

1. fs.stat(path, callback)

**Parameter explanation:**

**Path:**This is string having file name including path.

**Callback:**This is the callback function which gets two arguments (err, stats) where stats is an object of fs.Stats type.

Node.js fs.Stats class Methods

|  |  |
| --- | --- |
| **Method** | **Description** |
| stats.isfile() | returns true if file type of a simple file. |
| stats.isdirectory() | returns true if file type of a directory. |
| stats.isblockdevice() | returns true if file type of a block device. |
| stats.ischaracterdevice() | returns true if file type of a character device. |
| stats.issymboliclink() | returns true if file type of a symbolic link. |
| stats.isfifo() | returns true if file type of a fifo. |
| stats.issocket() | returns true if file type of asocket. |

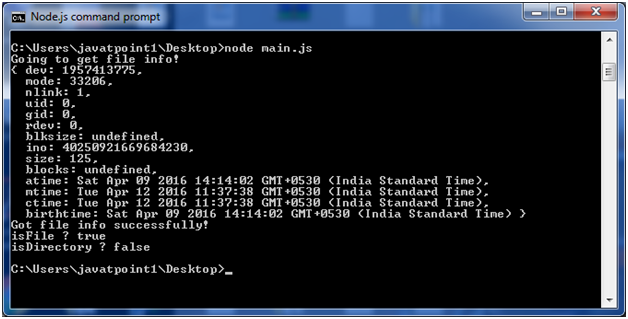
Let's take an example to create a JavaScript file named main.js having the following code:

*File: main.js*

1. var fs = require("fs");
2. console.log("Going to get file info!");
3. fs.stat('input.txt', function (err, stats) {
4. if (err) {
5. return console.error(err);
6. }
7. console.log(stats);
8. console.log("Got file info successfully!");
9. // Check file type
10. console.log("isFile ? " + stats.isFile());
11. console.log("isDirectory ? " + stats.isDirectory());
12. });

Now open the Node.js command prompt and run the main.js

1. node main.js



Node.js Path

The Node.js path module is used to handle and transform files paths. This module can be imported by using the following syntax:

**Syntax:**

1. var path =  require ("path")

Node.js Path Methods

Let's see the list of methods used in path module:

|  |  |  |
| --- | --- | --- |
| **Index** | **Method** | **Description** |
| 1. | path.normalize(p) | It is used to normalize a string path, taking care of '..' and '.' parts. |
| 2. | path.join([path1][, path2][, ...]) | It is used to join all arguments together and normalize the resulting path. |
| 3. | path.resolve([from ...], to) | It is used to resolve an absolute path. |
| 4. | path.isabsolute(path) | It determines whether path is an absolute path. an absolute path will always resolve to the same location, regardless of the working directory. |
| 5. | path.relative(from, to) | It is used to solve the relative path from "from" to "to". |
| 6. | path.dirname(p) | It return the directory name of a path. It is similar to the unix dirname command |
| 7. | path.basename(p[, ext]) | It returns the last portion of a path. It is similar to the Unix basename command. |
| 8. | path.extname(p) | It returns the extension of the path, from the last '.' to end of string in the last portion of the path. if there is no '.' in the last portion of the path or the first character of it is '.', then it returns an empty string. |
| 9. | path.parse(pathstring) | It returns an object from a path string. |
| 10. | path.format(pathobject) | It returns a path string from an object, the opposite of path.parse above. |

Node.js Path Example

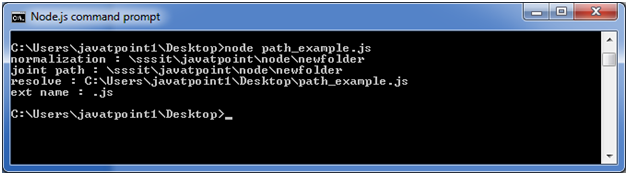
*File: path\_example.js*

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. var path = require("path");
2. // Normalization
3. console.log('normalization : ' + path.normalize('/sssit/javatpoint//node/newfolder/tab/..'));
4. // Join
5. console.log('joint path : ' + path.join('/sssit', 'javatpoint', 'node/newfolder', 'tab', '..'));
6. // Resolve
7. console.log('resolve : ' + path.resolve('path\_example.js'));
8. // Extension
9. console.log('ext name: ' + path.extname('path\_example.js'));

Open Node.js command prompt and run the path\_example.js:

1. node path\_example.js



Node.js StringDecoder

The Node.js StringDecoder is used to decode buffer into string. It is similar to buffer.toString() but provides extra support to UTF.

You need to use require('string\_decoder') to use StringDecoder module.

1. **const** StringDecoder = require('string\_decoder').StringDecoder;

Node.js StringDecoder Methods

StringDecoder class has two methods only.

|  |  |
| --- | --- |
| **Method** | **Description** |
| decoder.write(buffer) | It is used to return the decoded string. |
| decoder.end() | It is used to return trailing bytes, if any left in the buffer. |

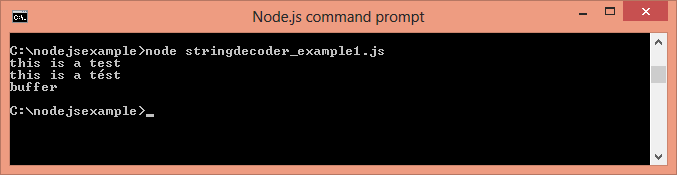
Node.js StringDecoder Example

Let's see a simple example of Node.js StringDecoder.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

*File: stringdecoder\_example1.js*

1. **const** StringDecoder = require('string\_decoder').StringDecoder;
2. **const** decoder = **new** StringDecoder('utf8');
4. **const** buf1 = **new** Buffer('this is a test');
5. console.log(decoder.write(buf1));//prints: this is a test
7. **const** buf2 = **new** Buffer('7468697320697320612074c3a97374', 'hex');
8. console.log(decoder.write(buf2));//prints: this is a test
10. **const** buf3 = Buffer.from([0x62,0x75,0x66,0x66,0x65,0x72]);
11. console.log(decoder.write(buf3));//prints: buffer



Next Topic[Node.js Query String](https://www.javatpoint.com/nodejs-query-string)

Node.js Query String

The Node.js Query String provides methods to deal with query string. It can be used to convert query string into JSON object and vice-versa.

To use query string module, you need to use **require('querystring')**.

Node.js Query String Methods

The Node.js Query String utility has four methods. The two important methods are given below.

|  |  |
| --- | --- |
| **Method** | **Description** |
| querystring.parse(str[, sep][, eq][, options]) | converts query string into JSON object. |
| querystring.stringify(obj[, sep][, eq][, options]) | converts JSON object into query string. |

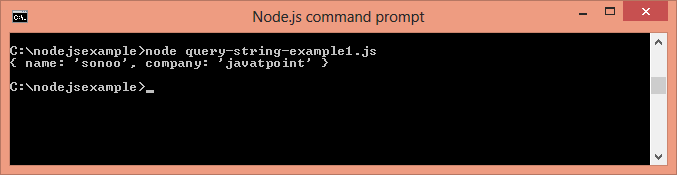
Node.js Query String Example 1: parse()

Let's see a simple example of Node.js Query String parse() method.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

*File: query-string-example1.js*

1. querystring = require('querystring');
2. **const** obj1=querystring.parse('name=sonoo&company=javatpoint');
3. console.log(obj1);

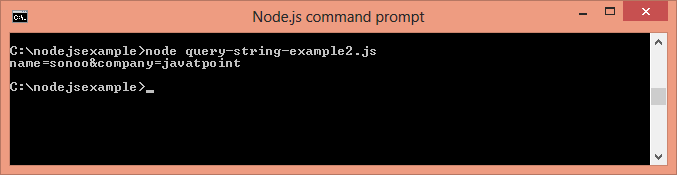


Node.js Query String Example 2: stringify()

Let's see a simple example of Node.js Query String stringify() method.

*File: query-string-example2.js*

1. querystring = require('querystring');
2. **const** qs1=querystring.stringify({name:'sonoo',company:'javatpoint'});
3. console.log(qs1);



# Node.js ZLIB

The Node.js Zlib module is used to provide compression and decompression (zip and unzip) functionalities. It is implemented using Gzip and deflate/inflate.

The zlib module can be accessed using:

1. const zlib = require('zlib');

Compressing and decompressing a file can be done by piping the source stream data into a destination stream through zlib stream.

## Node.js ZLIB Example: Compress File

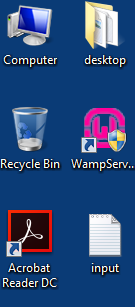
Let's see a simple example of Node.js ZLIB module to compress a file "input.txt" into "input.txt.gz".

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

*File: zlib\_example1.js*

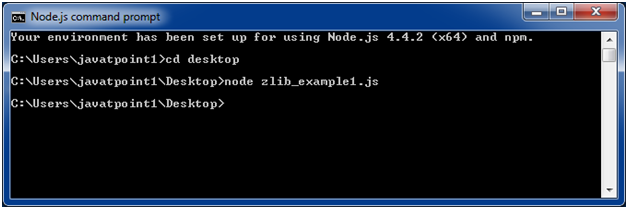
1. **const** zlib = require('zlib');
2. **const** gzip = zlib.createGzip();
3. **const** fs = require('fs');
4. **const** inp = fs.createReadStream('input.txt');
5. **const** out = fs.createWriteStream('input.txt.gz');
6. inp.pipe(gzip).pipe(out);

We have a text file named "input.txt" on the desktop.

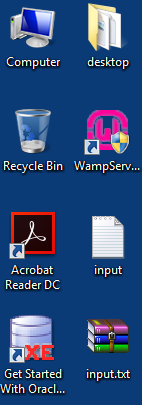


Open Node.js command prompt and run the following code:

1. node zlib\_example1.js



You can see that it will produce a compressed file named "input.txt.gz" on the desktop.



## Node.js ZLIB Example: Decompress File

Let's see a simple example of Node.js ZLIB module to decompress a file "input.txt.gz" into "input2.txt".

*File: zlib\_example2.js*

1. **const** zlib = require('zlib');
2. **const** unzip = zlib.createUnzip();
3. **const** fs = require('fs');
4. **const** inp = fs.createReadStream('input.txt.gz');
5. **const** out = fs.createWriteStream('input2.txt');
7. inp.pipe(unzip).pipe(out);
8. node zlib\_example2.js

Now you will see that same code of "input.txt" is available into "input2.txt" file.

#### **To understand this example well, create "input.txt" file having a large amount of data. Let's assume it has 40 kb data. After compressing this file you will get the size of compressed file "input.txt.gz" to 1 kb only. After decompressing the "input.txt.gz" file, you will get 40 kb of same data into "input2.txt" file.**

Node.js Assertion Testing

The Node.js Assert is the most elementary way to write tests. It provides no feedback when running your test unless one fails. The assert module provides a simple set of assertion tests that can be used to test invariants. The module is intended for internal use by Node.js, but can be used in application code via require ('assert').

However assert is not a testing framework and cannot be used as general purpose assertion library.

Node.js Assert Example

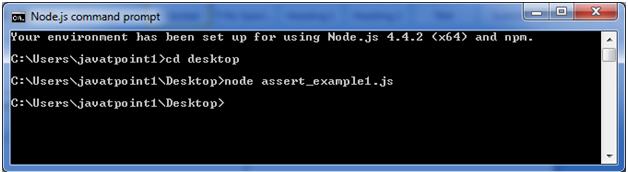
Let's see a simple example of Node.js Assert.

*File: assert\_example1.js*

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. var **assert** = require('assert');
2. function add (a, b) {
3. **return** a + b;
4. }
5. var expected = add(1,2);
6. **assert**( expected === 3, 'one plus two is three');

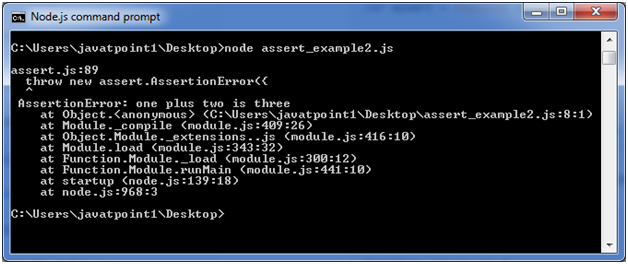
It will not provide any output because the case is true. If you want to see output, you need to make the test fail.



*File: assert\_example2.js*

1. var **assert** = require('assert');
2. function add (a, b) {
3. **return** a + b;
4. }
5. var expected = add(1,2);
6. **assert**( expected === 4, 'one plus two is three');

Now you will see the AssertionError.



Node.js V8

What is V8

V8 is an open source JavaScript engine developed by the Chromium project for the Google Chrome web browser. It is written in C++. Now a days, it is used in many projects such as Couchbase, MongoDB and Node.js.

V8 in Node.js

The Node.js V8 module represents interfaces and event specific to the version of V8. It provides methods to get information about heap memory through **v8.getHeapStatistics()** and **v8.getHeapSpaceStatistics()** methods.

To use this module, you need to use **require('v8')**.

1. **const** v8 = require('v8');

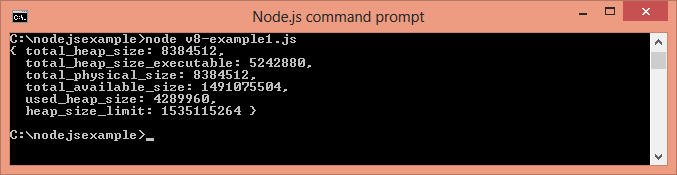
Node.js v8.getHeapStatistics() Example

The v8.getHeapStatistics() method returns statistics about heap such as total heap size, used heap size, heap size limit, total available size etc.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

*File: v8-example1.js*

1. **const** v8 = require('v8');
2. console.log(v8.getHeapStatistics());

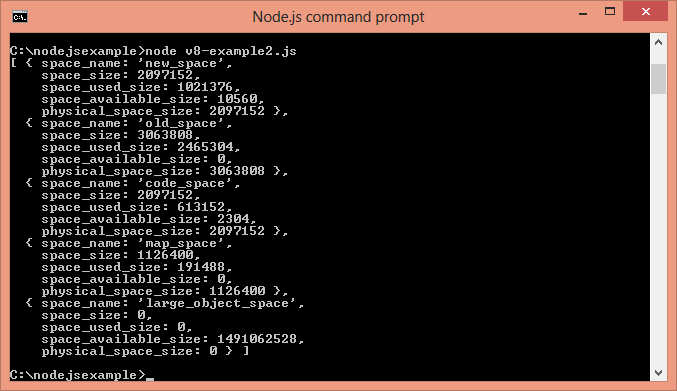


Node.js v8.getHeapSpaceStatistics() Example

The v8.getHeapSpaceStatistics() returns statistics about heap space. It returns an array of 5 objects: new space, old space, code space, map space and large object space. Each object contains information about space name, space size, space used size, space available size and physical space size.

*File: v8-example2.js*

1. **const** v8 = require('v8');
2. console.log(v8.getHeapSpaceStatistics());



Memory limit of V8 in Node.js

Currently, by default v8 has a memory limit of 512mb on 32-bit and 1gb on 64-bit systems. You can raise the limit by setting --max-old-space-size to a maximum of ~1gb for 32-bit and ~1.7gb for 64-bit systems. But it is recommended to split your single process into several workers if you are hitting memory limits.

# Node.js Callbacks

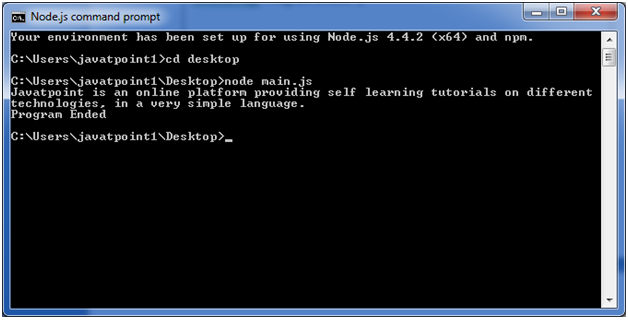
Callback is an asynchronous equivalent for a function. It is called at the completion of each task. In Node.js, callbacks are generally used. All APIs of Node are written in a way to supports callbacks. For example: when a function start reading file, it returns the control to execution environment immediately so that the next instruction can be executed.

In Node.js, once file I/O is complete, it will call the callback function. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process high number of request without waiting for any function to return result.

## Blocking Code Example

Follow these steps:

1. Create a text file named **input.txt**having the following content:
   1. Javatpoint is an online platform providing self learning tutorials on
   2. different technologies, in a very simple language.
2. Create a JavaScript file named **main.js** having the following code:
   1. var fs = require("fs");
   2. var data = fs.readFileSync('input.txt');
   3. console.log(data.toString());
   4. console.log("Program Ended");
3. Open the **Node.js** command prompt and execute the following code.
   1. node main.js

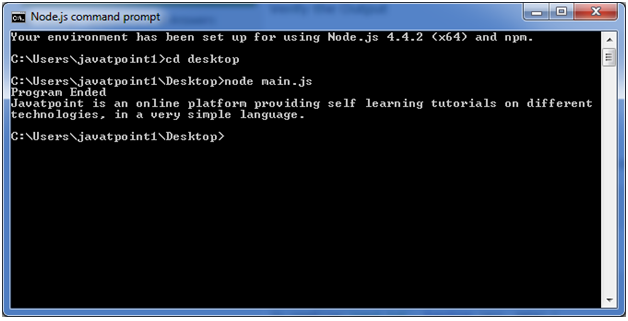


## Non Blocking Code Example

Follow these steps:

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. Create a text file named **input.txt**having the following content:
   1. Javatpoint is an online platform providing self learning tutorials on
   2. different technologies, in a very simple language.
2. Create a JavaScript file named **main.js** having the following code:
   1. var fs = require("fs");
   3. fs.readFile('input.txt', function (err, data) {
   4. if (err) return console.error(err);
   5. console.log(data.toString());
   6. });
   7. console.log("Program Ended");
3. Open the **Node.js** command prompt and execute the following code.
   1. node main.js



#### **You can see that above two examples explain the concept of blocking and non-blocking calls. The first example shows that program blocks until it reads the file and then only it proceeds to end the program on the other hand in second example, program does not wait for file reading but it just proceeded to print "Program Ended" and same time program without blocking continues reading the file.**

So we can say that, a blocking program executes very much in sequence. It is also easier to implement the logic from programming point of view in block programs. But non-blocking programs does not execute in sequence, so in case a program needs to use any data to be processed, it should be kept with-in the same block to make it sequential execution.

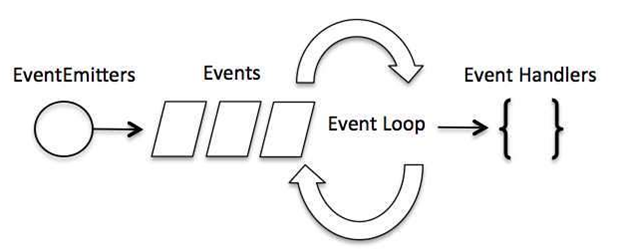
Node.js Events

In Node.js applications, Events and Callbacks concepts are used to provide concurrency. As Node.js applications are single threaded and every API of Node js are asynchronous. So it uses async function to maintain the concurrency. Node uses observer pattern. Node thread keeps an event loop and after the completion of any task, it fires the corresponding event which signals the event listener function to get executed.

Event Driven Programming

Node.js uses event driven programming. It means as soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for event to occur. It is the one of the reason why Node.js is pretty fast compared to other similar technologies.

There is a main loop in the event driven application that listens for events, and then triggers a callback function when one of those events is detected.



Difference between Events and Callbacks:

Although, Events and Callbacks look similar but the differences lies in the fact that callback functions are called when an asynchronous function returns its result where as event handling works on the observer pattern. Whenever an event gets fired, its listener function starts executing. Node.js has multiple in-built events available through events module and EventEmitter class which is used to bind events and event listeners.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

**EventEmitter class to bind event and event listener:**

1. // Import events module
2. var events = require('events');
3. // Create an eventEmitter object
4. var eventEmitter = new events.EventEmitter();

**To bind event handler with an event:**

1. // Bind event and even handler as follows
2. eventEmitter.on('eventName', eventHandler);

**To fire an event:**

1. // Fire an event
2. eventEmitter.emit('eventName');

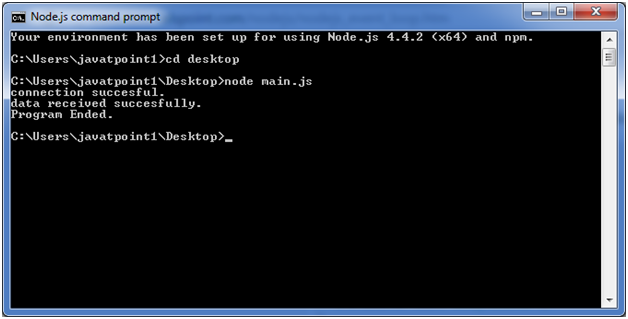
Node.js Event Example

*File: main.js*

1. // Import events module
2. var events = require('events');
3. // Create an eventEmitter object
4. var eventEmitter = new events.EventEmitter();
6. // Create an event handler as follows
7. var connectHandler = function connected() {
8. console.log('connection succesful.');
10. // Fire the data\_received event
11. eventEmitter.emit('data\_received');
12. }
14. // Bind the connection event with the handler
15. eventEmitter.on('connection', connectHandler);
16. // Bind the data\_received event with the anonymous function
17. eventEmitter.on('data\_received', function(){
18. console.log('data received succesfully.');
19. });
20. // Fire the connection event
21. eventEmitter.emit('connection');
22. console.log("Program Ended.");

Now, open the Node.js command prompt and run the following code:

1. node main.js



Node.js Punycode

What is Punycode

Punycode is an encoding syntax which is used to convert Unicode (UTF-8) string of characters to basic ASCII string of characters. Since host names only understand ASCII characters so Punycode is used. It is used as an internationalized domain name (IDN or IDNA). Let's understand it with an example:

Assume if you search for **mañana.com** in your browser so your browser (which is IDNA enabled) first convert this to punycode xn--maana-pta.com because the character **ñ** is not allowed in regular domain name. It is not supported in older versions.

Punycode in Node.js

Punycode.js is bundled with Node.js v0.6.2 and later versions. If you want to use it with other Node.js versions then use npm to install punycode module first. You have to use require ('punycode') to access it.

**Syntax:**

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

1. punycode = require('punycode');

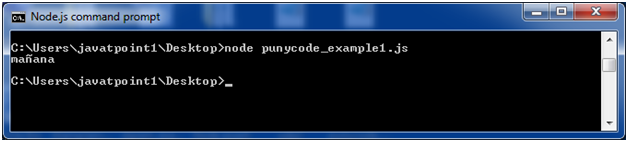
punycode.decode(string)

It is used to convert a Punycode string of ASCII symbols to a string of Unicode symbols.

*File: punycode\_example1.js*

1. punycode = require('punycode');
2. console.log(punycode.decode('maana-pta'));

**Output:**



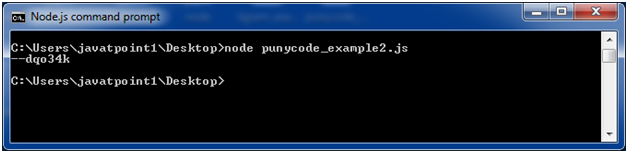
punycode.encode(string)

It is used to convert a string of Unicode symbols to a Punycode string of ASCII symbols.

*File: punycode\_example2.js*

1. punycode = require('punycode');
2. console.log(punycode.encode('☃-⌘'));

**Output:**



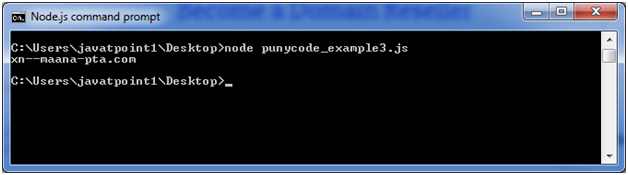
punycode.toASCII(domain)

It is used to convert a Unicode string representing a domain name to Punycode. Only the non-ASCII part of the domain name is converted.

*File: punycode\_example3.js*

1. punycode = require('punycode');
2. console.log(punycode.toASCII('mañana.com'));

**Output:**



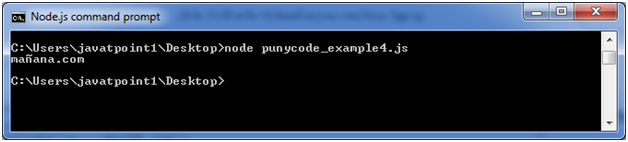
punycode.toUnicode(domain)

It is used to convert a Punycode string representing a domain name to Unicode. Only the Punycoded part of the domain name is converted.

*File: punycode\_example4.js*

1. punycode = require('punycode');
2. console.log(punycode.toUnicode('xn--maana-pta.com'));

**Output:**



Node.js TTY

The Node.js TTY module contains tty.ReadStream and tty.WriteStream classes. In most cases, there is no need to use this module directly.

You have to use require ('tty') to access this module.

**Syntax:**

1. var tty = require('tty');

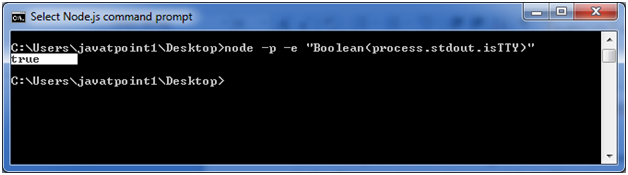
When Node.js discovers that it is being run inside a TTY context, then:

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

* process.stdin will be a tty.ReadStream instance
* process.stdout will be a tty.WriteStream instance

To check that if Node.js is running in a TTY context, use the following command:

1. node -p -e "Boolean(process.stdout.isTTY)"



Class: ReadStream

It contains a net.Socket subclass that represents the readable portion of a tty. In normal circumstances,the tty.ReadStream has the only instance named process.stdin in any Node.js program (only when isatty(0) is true).

**rs.isRaw:** It is a Boolean that is initialized to false. It specifies the current "raw" state of the tty.ReadStream instance.

**rs.setRawMode(mode):** It should be true or false. It is used to set the properties of the tty.ReadStream to act either as a raw device or default. isRaw will be set to the resulting mode.

Class: WriteStream

It contains a net.Socket subclass that represents the writable portion of a tty. In normal circumstances, the tty.WriteStream has the only instance named process.stdout in any Node.js program (only when isatty(1) is true).

**Resize event:** This event is used when either of the columns or rows properties has changed.

**Syntax:**

1. process.stdout.on('resize', () =**>** {
2. console.log('screen size has changed!');
3. console.log(`${process.stdout.columns}x${process.stdout.rows}`);
4. });

**ws.columns:** It is used to give the number of columns the TTY currently has. This property gets updated on 'resize' events.

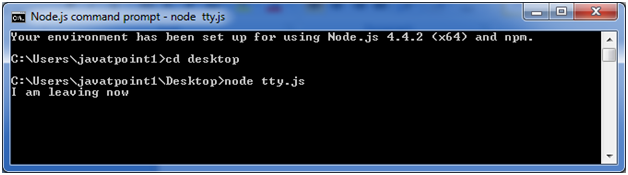
**ws.rows:** It is used to give the number of rows the TTY currently has. This property gets updated on 'resize' events.

Node.js TTY Example

*File: tty.js*

1. var tty = require('tty');
2. process.stdin.setRawMode(**true**);
3. process.stdin.resume();
4. console.log('I am leaving now');
5. process.stdin.on('keypress', function(**char**, key) {
6. **if** (key && key.ctrl && key.name == 'c') {
8. process.exit()
9. }
10. });

**Output:**



Node.js Web Module

What is Web Server

Web Server is a software program that handles HTTTP requests sent by HTTP clients like web browsers, and returns web pages in response to the clients. Web servers usually respond with html documents along with images, style sheets and scripts.

Most of the web server support server side scripts using scripting language or redirect to application server which perform the specific task of getting data from database, perform complex logic etc. and then sends a result to the HTTP client through the Web server.

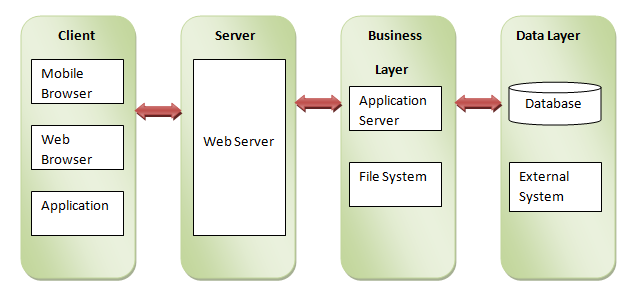
Apache web server is one of the most commonly used web server. It is an open source project.

Web Application Architecture

A web application can be divided in 4 layers:

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

* **Client Layer:**The Client layer contains web browsers, mobile browsers or applications which can make HTTP request to the web server.
* **Server Layer:**The Server layer contains Web server which can intercepts the request made by clients and pass them the response.
* **Business Layer:**The business layer contains application server which is utilized by web server to do required processing. This layer interacts with data layer via data base or some external programs.
* **Data Layer:**The Data layer contains databases or any source of data.



Creating Web Server using Node.js

Node.js provides http module which can be used to create either HTTP client of server. Create a js file named server.js having the following code:

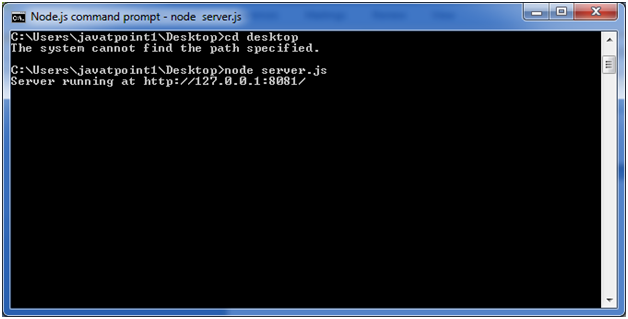
1. var http = require('http');
2. var fs = require('fs');
3. var url = require('url');
5. // Create a server
6. http.createServer( function (request, response) {
7. // Parse the request containing file name
8. var pathname = url.parse(request.url).pathname;
10. // Print the name of the file for which request is made.
11. console.log("Request for " + pathname + " received.");
13. // Read the requested file content from file system
14. fs.readFile(pathname.substr(1), function (err, data) {
15. if (err) {
16. console.log(err);
17. // HTTP Status: 404 : NOT FOUND
18. // Content Type: text/plain
19. response.writeHead(404, {'Content-Type': 'text/html'});
20. }else{
21. //Page found
22. // HTTP Status: 200 : OK
23. // Content Type: text/plain
24. response.writeHead(200, {'Content-Type': 'text/html'});
26. // Write the content of the file to response body
27. response.write(data.toString());
28. }
29. // Send the response body
30. response.end();
31. });
32. }).listen(8081);
33. // Console will print the message
34. console.log('Server running at http://127.0.0.1:8081/');

**Next, create an html file named index.html having the following code in the same directory where you created server.js**

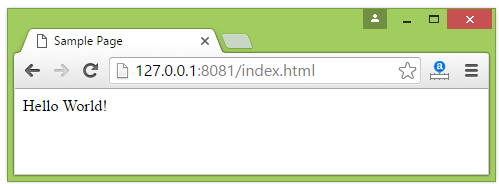
1. **<html>**
2. **<head>**
3. **<title>**Sample Page**</title>**
4. **</head>**
5. **<body>**
6. Hello World!
7. **</body>**
8. **</html>**

Now open the Node.js command prompt and run the following code:

**node server.js**



Open http://127.0.0.1:8081/index.htm in any browser and see the below result.



# Node.js Interview Questions

A list of top frequently asked **Node.js interview questions** and answers are given below.

### **1) What is Node.js?**

Node.js is Server-side scripting which is used to build scalable programs. It is a web application framework built on Google Chrome's JavaScript Engine. It runs within the Node.js runtime on Mac OS, Windows, and Linux with no changes. This runtime facilitates you to execute a JavaScript code on any machine outside a browser.

### **2) Is Node.js free to use?**

Yes. It is released under MIT license and is free to use.

### **3) Is Node a single threaded application?**

Yes. Node is a single-threaded application with event looping.

Play Video[](https://campaign.adpushup.com/get-started/?utm_source=banner&utm_campaign=growth_hack)

### **4) What is the purpose of Node.js?**

These are the following purposes of Node.js:

* Real-time web applications
* Network applications
* Distributed systems
* General purpose applications

### **5) What are the advantages of Node.js?**

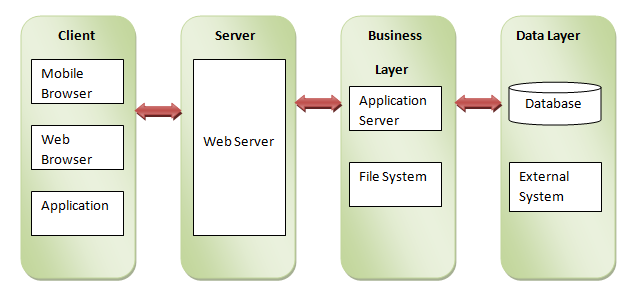
Following are the main advantages of Node.js:

* Node.js is asynchronous and event-driven. All API?s of Node.js library are non-blocking, and its server doesn't wait for an API to return data. It moves to the next API after calling it, and a notification mechanism of Events of Node.js responds to the server from the previous API call.
* Node.js is very fast because it builds on Google Chrome?s V8 JavaScript engine. Its library is very fast in code execution.
* Node.js is single threaded but highly scalable.
* Node.js provides a facility of no buffering. Its application never buffers any data. It outputs the data in chunks.

### **6) Explain Node.js web application architecture?**

A web application distinguishes into 4 layers:

* **Client Layer:** The Client layer contains web browsers, mobile browsers or applications which can make an HTTP request to the web server.
* **Server Layer:** The Server layer contains the Web server which can intercept the request made by clients and pass them the response.
* **Business Layer:** The business layer contains application server which is utilized by the web server to do required processing. This layer interacts with the data layer via database or some external programs.
* **Data Layer:** The Data layer contains databases or any source of data.



### **7) What do you understand by the term I/O?**

The term I/O stands for input and output. It is used to access anything outside of your application. The I/O describes any program, operation, or device that transfers data to or from a medium or another medium. This medium can be a physical device, network, or files within a system.

I/O is loaded into the machine memory to run the program once the application starts.

### **8) How many types of API functions are available in Node.js?**

There are two types of API functions in Node.js:

* Asynchronous, Non-blocking functions
* Synchronous, Blocking functions

### **9) What do you understand by the first class function in JavaScript?**

When functions are treated like any other variable, then those functions are called first-class functions. Apart from JavaScript, many other programming languages, such as Scala, Haskell, etc. follow this pattern. The first class functions can be passed as a param to another function (callback), or a function can return another function (higher-order function). Some examples of higher-order functions that are popularly used are map() and filter().

### **10) What is the difference between JavaScript and Node.js?**

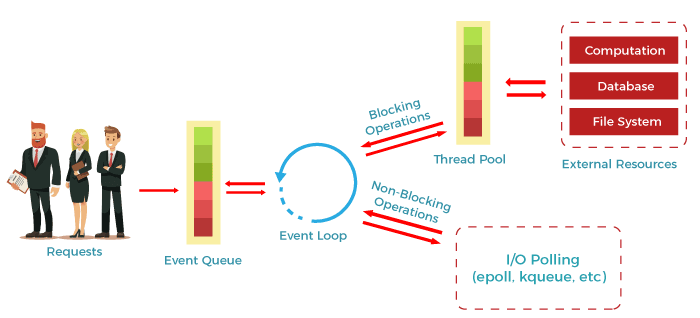
**Difference between JavaScript and Node.js**

The following table specifies the crucial differences between JavaScript and Node.js:

|  |  |  |
| --- | --- | --- |
| **Comparison features** | **JavaScript** | **Node.js** |
| Type | JavaScript is a programming language. More precisely, you can say that it is a scripting language used for writing scripts on the website. | Node.js is an interpreter and run time environment for JavaScript. |
| Utility | JavaScript is used for any client-side activity for a web application. | Node.js is used for accessing or performing any non-blocking operation of any operating system. |
| Running Engine | The running engine for JavaScript is Spider monkey (Firefox), JavaScript Core (Safari), V8 (Google Chrome), etc. | The running engine for Node.js is V8 (Google Chrome). |
| Browser compatibility | JavaScript can only be run in browsers. | The Node.js code can be run outside the browser. |
| Platform dependency | JavaScript is basically used on the client-side and is used in frontend development. | Node.js is mostly used on the server-side and is used in server-side development. |
| HTML compatibility | JavaScript is capable enough to add HTML and play with the DOM. | Node.js is not compatible enough to add HTML tags. |
| Examples | Some examples of the JavaScript frameworks are RamdaJS, TypedJS, etc. | Some examples of the Node.js modules are Lodash, express, etc. We have to import these modules from npm. |
| Written in | JavaScript is the upgraded version of ECMA script that uses Chrome's V8 engine and is written in C++. | Node.js is written in C, C++, and Javascript. |

### **11) Explain the working of Node.js?**

The workflow of a Node.js web server typically looks like the following diagram. Let us see the flow of operations in detail:



* According to the above diagram, the clients send requests to the webserver to interact with the web application. These requests can be non-blocking or blocking and used for querying the data, deleting data, or updating the data.
* js receives the incoming requests and adds those to the Event Queue.
* After this step, the requests are passed one by one through the Event Loop. It checks if the requests are simple enough not to require any external resources.
* The event loop then processes the simple requests (non-blocking operations), such as I/O Polling, and returns the responses to the corresponding clients.
* A single thread from the Thread Pool is assigned to a single complex request. This thread is responsible for completing a particular blocking request by accessing external resources, such as computation, database, file system, etc.
* Once the task is completed, the response is sent to the Event Loop that sends that response back to the client.

### **12) How can you manage the packages in your Node.js project?**

We can manage the packages in our Node.js project by using several package installers and their configuration file accordingly. Most of them use npm or yarn. The npm and yarn both provide almost all libraries of JavaScript with extended features of controlling environment-specific configurations. We can use package.json and package-lock.json to maintain versions of libs being installed in a project. So, there is no issue in porting that app to a different environment.

### **13) Why is Node.js Single-threaded?**

Node.js is a single-threaded application with event looping for async processing. The biggest advantage of doing async processing on a single thread under typical web loads is that you can achieve more performance and scalability than the typical thread-based implementation.

### **14) What do you understand by callback hell in Node.js?**

Callback hell is a phenomenon that creates a lot of problems for a JavaScript developer when he tries to execute multiple asynchronous operations one after the other. A function is called an asynchronous function when some external activity must complete before processing a result. It is called asynchronous because there is an unpredictable amount of time before a result becomes available. These functions require a callback function to handle errors and process the result.

**Example:**

1. getData(function(a){
2. getMoreData(a, function(b){
3. getMoreData(b, function(c){
4. getMoreData(c, function(d){
5. getMoreData(d, function(e){
6. ...
7. });
8. });
9. });
10. });
11. });

### **15) How is Node.js better than other most popular frameworks?**

Based on the following criteria, we can say that Node.js is better than other most popular frameworks:

* js makes development simple because of its non-blocking I/O and even-based model. This simplicity results in short response time and concurrent processing, unlike other frameworks where developers use thread management.
* js runs on a chrome V8 engine which is written in C++. It enhances its performance highly with constant improvement.
* With Node.js, we will use JavaScript in both the frontend and backend development that will be much faster.
* js provides ample libraries so that we don't need to reinvent the wheel.

### **16) In which types of applications is Node.js most frequently used?**

Node.js is most frequently and widely used in the following applications:

* Internet of Things
* Real-time collaboration tools
* Real-time chats
* Complex SPAs (Single-Page Applications)
* Streaming applications
* Microservices architecture etc.

### **17) What are some commonly used timing features of Node.js?**

Following is a list of some commonly used timing features of Node.js:

* **setTimeout/clearTimeout:** This timing feature of Node.js is used to implement delays in the code execution.
* **setInterval/clearInterval:** The setInterval or clearInterval timing feature is used to run a code block multiple times in the application.
* **setImmediate/clearImmediate:** This timing feature of Node.js is used to set the execution of the code at the end of the event loop cycle.
* **nextTick:** This timing feature sets the execution of code at the beginning of the next event loop cycle.

### **18) What do you understand by the term fork in Node.js?**

Generally, a fork is used to spawn child processes. In Node.js, it is used to create a new instance of the V8 engine to run multiple workers to execute the code.

### **19) Which is the best tool we can use to assure consistent code style in Node.js?**

ESLint tool is one of the best tools we can use with any IDE to ensure a consistent coding style. It also helps in maintaining the codebase.

### **20) What is the main difference between front-end and back-end development?**

The following table specifies the key differences between a front-end and back-end development:

|  |  |
| --- | --- |
| **Front-end Development** | **Back-end Development** |
| The front-end development in an application refers to the client-side of an application. | The back-end development in an application refers to the server-side of an application. |
| As the name specifies, the front-end development is the part of a web application where users can see and interact. | As the name specifies, the back-end development consists of everything that happens behind the scenes and users cannot see and interact with. |
| The front-end development includes everything that attributes to the visual aspects of a web application. | The back-end development generally includes a web server that communicates with the database to serve the users' requests. |
| HTML, CSS, Bootstrap, jQuery, JavaScript, AngularJS, and React.js are essential front-end development technologies. | Java, PHP, Python, C++, Node.js, etc., are the technologies required for back-end development. |
| Examples of some front-end frameworks are AngularJS, React.js, jQuery, Sass, etc. | Examples of some back-end frameworks are Express, Django, Rails, Laravel, Spring, etc. |

### **21) Give an example to demonstrate how can we use async await in Node.js?**

See the following example of using async-await pattern:

1. function wait (timeout) {
2. **return** **new** Promise((resolve) => {
3. setTimeout(() => {
4. resolve()
5. }, timeout);
6. });
7. }
8. async function requestWithRetry (url) {
9. **const** MAX\_RETRIES = 10;
10. **for** (let i = 0; i <= MAX\_RETRIES; i++) {
11. **try** {
12. **return** await request(url);
13. } **catch** (err) {
14. **const** timeout = Math.pow(2, i);
15. console.log('Waiting', timeout, 'ms');
16. await wait(timeout);
17. console.log('Retrying', err.message, i);
18. }
19. }
20. }

### **22) What are the modules in Node.js? Which are the different modules used in Node.js?**

In Node.js applications, modules are like JavaScript libraries and include a set of functions. To include a module in a Node.js application, we must use the require() function with the parentheses containing the module's name.

Node.js has several modules which are used to provide the basic functionality needed for a web application. Following is a list of some of them:

|  |  |
| --- | --- |
| **Core Modules** | **Description** |
| HTTP: | The HTTP module includes classes, methods, and events to create a Node.js HTTP server. |
| util: | The util module includes utility functions required in the application and is very useful for developers. |
| url: | The url module is used to include the methods for URL parsing. |
| fs: | The fs module includes events, classes, and methods to handle the file I/O operations. |
| stream: | The stream module is used to include the methods to handle streaming data. |
| query string: | The query string module is used to include the methods to work with a query string. |
| zlib: | The zlib module is used to include the methods to compress or decompress the files used in an application. |

### **23) What are buffers in Node.js?**

In general, a buffer is a temporary memory mainly used by the stream to hold on to some data until it is consumed. Buffers are used to represent a fixed-size chunk of memory allocated outside of the V8 JavaScript engine. It can't be resized. It is like an array of integers, which each represents a byte of data. It is implemented by the Node. js Buffer class. Buffers also support legacy encodings like ASCII, utf-8, etc.

### **24) What is error-first callback?**

Error-first callbacks are used to pass errors and data. If something goes wrong, the programmer has to check the first argument because it is always an error argument. Additional arguments are used to pass data.

1. fs.readFile(filePath, function(err, data) {
2. if (err) {
3. //handle the error
4. }
5. // use the data object
6. });

### **25) What is an asynchronous API?**

All the API's of Node.js library are asynchronous means non-blocking. A Node.js based server never waits for an API to return data. The Node.js server moves to the next API after calling it, and a notification mechanism of Events of Node.js responds to the server for the previous API call.

### **26) How can you avoid callbacks?**

To avoid callbacks, you can use any one of the following options:

* You can use **modularization**. It breaks callbacks into independent functions.
* You can use **promises**.
* You can use **yield** with Generators and Promises.

### **27) Does Node.js provide Debugger?**

Yes, Node.js provides a simple TCP based protocol and built-in debugging client. For debugging your JavaScript file, you can use debug argument followed by the js file name you want to debug.

#### **Syntax:**

1. node debug [script.js | -e "script" | **<host>**:**<port>**]

### **28) What is a control flow function?**

Control flow function is a generic piece of code that runs in between several asynchronous function calls.

### **29) How "Control Flow" controls the functions calls?**

The control flow does the following job:

* Control the order of execution
* Collect data
* Limit concurrency
* Call the next step in a program

### **30) Is it possible to access DOM in Node?**

No, it is not possible to access DOM in Node.

### **31) What types of tasks can be done asynchronously using the event loop?**

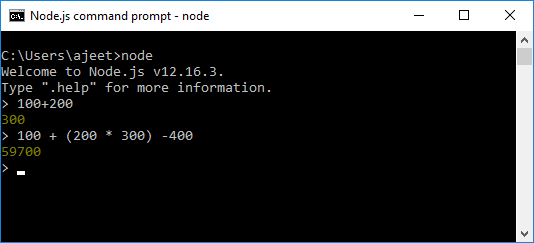
* I/O operations
* Heavy computation
* Anything requiring blocking

### **32) What is REPL in Node.js?**

REPL stands for Read Eval Print Loop. It specifies a computer environment like a window console or Unix/Linux shell where you can enter a command, and the computer responds with an output. It is very useful in writing and debugging the codes. REPL environment incorporates Node.js.

**See the Example:**

1. $ node
2. > 100 + 200
3. 300
4. > 100 + ( 200 \* 300 ) - 400
5. 59700
6. >



### **33) Explain the tasks of terms used in Node REPL.**

Following are the terms used in REPL with their defined tasks:

**Read:** It reads user's input; parse the input into JavaScript data-structure and stores in memory.

**Eval:** It takes and evaluates the data structure.

**Print:** It is used to print the result.

**Loop:** It loops the above command until user press ctrl-c twice to terminate.

### **34) Is it possible to evaluate simple expressions using Node REPL?**

Yes. You can evaluate simple expressions using Node REPL.

### **35) What is the use of the underscore variable in REPL?**

In REPL, the underscore variable is used to get the last result.

1. C:\Nodejs\_WorkSpace**>**node
2. **>** var x = 10
3. undefined
4. **>** var y = 20
5. undefined
6. **>** x + y
7. 30
8. **>** var sum = \_
9. undefined
10. **>** console.log(sum)
11. 30
12. undefined
13. **>**

### **36) Does Node.js supports cryptography?**

Yes, Node.js Crypto module supports cryptography. It provides cryptographic functionality that includes a set of wrappers for open SSL's hash HMAC, cipher, decipher, sign and verify functions. For example:

1. const crypto = require('crypto');
2. const secret = 'abcdefg';
3. const hash = crypto.createHmac('sha256', secret)
4. .update('Welcome to JavaTpoint')
5. .digest('hex');
6. console.log(hash);

### **37) What is npm? What is the main functionality of npm?**

npm stands for Node Package Manager. Following are the two main functionalities of npm:

* Online repositories for node.js packages/modules which are searchable on search.nodejs.org
* Command line utility to install packages, do version management and dependency management of Node.js packages.

### **38) What tools can be used to assure a consistent style in Node.js?**

Following is a list of tools that can be used in developing code in teams, to enforce a given style guide and to catch common errors using static analysis.

* JSLint
* JSHint
* ESLint
* JSCS

### **39) What is the difference between operational and programmer errors?**

Operational errors are not bugs, but create problems with the system like request timeout or hardware failure. On the other hand, programmer errors are actual bugs.

### **40) What is the difference between the global installation of dependencies and local installation of dependencies?**

* Global installation of dependencies is stored in /npm directory. While local installation of dependencies stores in the local mode. Here local mode refers to the package installation in node\_modules directory lying in the folder where Node application is present.
* Globally deployed packages cannot be imported using require() in Node application directly. On the other hand, locally deployed packages are accessible via require().
* To install a Node project globally -g flag is used.
  1. C:\Nodejs\_WorkSpace**>**npm install express ?g
* To install a Node project locally, the syntax is:
  1. C:\Nodejs\_WorkSpace**>**npm install express

### **41) What is the use of a buffer class in Node.js?**

The Node.js provides Buffer class to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap. It is a global class and can be accessed in an application without importing a buffer module. Buffer class is used because pure JavaScript is not compatible with binary data. So, when dealing with TCP streams or the file system, it's necessary to handle octet streams.

### **42) What is the role of assert in Node.js?**

The Node.js Assert is a way to write tests. It provides no feedback when running your test unless one fails. The assert module provides a simple set of assertion tests that can be used to test invariants. The module is intended for internal use by Node.js, but can be used in application code via require ('assert'). For example:

1. var assert = require('assert');
2. function add (a, b) {
3. return a + b;
4. }
5. var expected = add(1,2);
6. assert( expected === 3, 'one plus two is three');

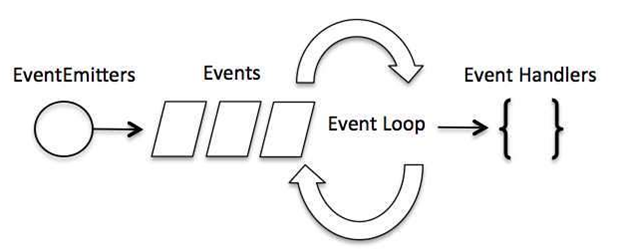
### **43) What are the streams in Node.js?**

The Streams are the objects that facilitate you to read data from a source and write data to a destination. There are four types of streams in Node.js:

* **Readable:** This stream is used for reading operations.
* **Writable:** This stream is used for write operations.
* **Duplex:** This stream can be used for both reading and write operations.
* **Transform:** It is a type of duplex stream where the output computes according to input.

### **44) What is event-driven programming in Node.js?**

In Node.js, event-driven programming means as soon as Node starts its server, it initiates its variables, declares functions and then waits for an event to occur. It is one of the reasons why Node.js is pretty fast compared to other similar technologies.



### **45) What is the difference between events and callbacks in Node.js?**

Although, Events and Callbacks look similar the differences lies in the fact that callback functions are called when an asynchronous function returns its result whereas event handling works on the observer pattern. Whenever an event gets fired, its listener function starts executing. Node.js has multiple in-built events available through the events module and EventEmitter class which is used to bind events and event listeners.

### **46) What is the Punycode in Node.js?**

The Punycode is an encoding syntax which is used to convert Unicode (UTF-8) string of characters to ASCII string of characters. It is bundled with Node.js v0.6.2 and later versions. If you want to use it with other Node.js versions, then use npm to install Punycode module first. You have to used require ('Punycode') to access it.

#### **Syntax:**

1. punycode = require('punycode');

### **47) What does Node.js TTY module contains?**

The Node.js TTY module contains tty.ReadStream and tty.WriteStream classes. In most cases, there is no need to use this module directly. You have to used require ('tty') to access this module.

#### **Syntax:**

1. var tty = require('tty');

### **48) What are the key differences between Angular and Node.js?**

Key differences between Angular and Node.js:

|  |  |
| --- | --- |
| **Angular** | **Node.js** |
| Angular is a structural front-end development framework for developing dynamic web apps. | Node.js is a cross-platform, run-time, server-side environment for applications written in JavaScript language. |
| Angular is entirely written in TypeScript language. | Node.js is written in C, C++, and JavaScript languages. |
| Angular is used for building single-page, client-side web applications. | Node.js is used for building fast and scalable, client-side, and server-side networking applications. |
| Angular is easy to use. The developers need to add the Angular file to use it in their applications. | Node.js is slightly complicated to use. Here, the developers need to install Node.js on their computer system. |
| Angular split a web application into MVC components. Here, the models and views are much simpler than what is found in other JavaScript client-side frameworks. | Node.js generates database queries and uses the event-driven nature of JavaScript to support non-blocking operations, making the platform efficient. |
| Angular is based on the model-view-controller design pattern and follows that pattern completely. | Node.js is single-threaded. It means the web requests and processing runs on the same thread. |
| Angular is a Web Framework. | Node.js provides different Web Frameworks like Socket.io, Hapi.js, Meteor.js, Express.js, and Sails.js, etc. |
| Angular is ideal for creating highly active and interactive web apps. | Node.js is the best for developing small-size projects. |
| Angular requires a deep understanding of prototyping, scope, and various other JavaScript aspects. | Node.js facilitates developers to use JavaScript on the client as well as the server-side. So, they can focus on learning one language. |

### **49) What are the main differences between operational and programmer errors?**

The most crucial difference between operational and programmer errors is that the operational errors are not bugs but problems with the system such as to request timeout or hardware failure. On the other hand, the programmer errors are actual bugs in the application.

### **50) What do you understand by an EventEmitter in Node.js?**

In Node.js, an EventEmitter is a class that includes all the objects capable of emitting events. This can be achieved by attaching named events that are emitted by the object using an eventEmitter.on() function. Thus whenever this object throws an event, the attached functions are invoked synchronously.

**Example:**

1. const EventEmitter = require('events');
2. class MyEmitter extends EventEmitter {}
3. const myEmitter = new MyEmitter();
4. myEmitter.on('event', () =**>** {
5. console.log('an event occurred!');
6. });
7. myEmitter.emit('event');

### **51) What is the difference between readFile and createReadStream in Node.js?**

In Node.js, there are two ways to read and execute files: readFile and CreateStream.

* The readFile() process is a fully buffered process that returns the response only when the complete file is pushed into the buffer and is read. This process is called a memory-intensive process, and in the case of large files, the processing can be very slow.
* On the other hand, the createReadStream() is a partially buffered process that treats the entire process as an event series. The entire file is split into chunks and then processed and sent back as a response one by one. After completing this step, they are finally removed from the buffer. Unlike the readFile process, the createReadStream process is effective for the processing of large files.

### **52) What is the concept of Punycode in Node.js?**

In Node.js, the concept of Punycode is used for converting one type of string into another type. Punycode is an encoding syntax used for converting Unicode (UTF-8) string of characters into a basic ASCII string of characters. Now, the hostnames can only understand the ASCII characters so, after the Node.js version 0.6.2 onwards, it was bundled up with the default Node package.

To use it with any previous versions, you have to use the following code:

**Syntax:**

1. punycode = require('punycode');

### **53) How can you enhance the Node.js performance through clustering?**

Just because the Node.js applications run on a single processor, they don't take advantage of a multiple-core system by default. Clustering is used to overcome this issue. The cluster mode is used to start up multiple node.js processes, thereby having multiple instances of the event loop. When we start using clusters in a Node.js app, it creates multiple node.js processes. But there is also a parent process called the cluster manager, which is responsible for monitoring the health of the individual instances of the application.



### **54) What is a thread pool in Node.js? Which library handles it?**

In Node.js, the libuv library is used to handle the Thread pool. The libuv library is a multi-platform C library that supports asynchronous I/O-based operations such as file systems, networking, and concurrency.

