INDEX

Sr. No.	Topic	Page No.
1.	Node.js Tutorial	1
2.	Install Node.js Windows	3
3.	Node.js First Example	4
4.	Node.js Console	6
5.	Node.js REPL	8
6.	Node.js NPM	11
7.	Node.js CL options	12
8.	Node.js Globals	15
9.	Node.js OS	16
10.	Node.js Timer	18
11.	Node.js Errors	20
12.	Node.js DNS	21
13.	Node.js Net	22
14.	Node.js Crypto	24
15.	Node.js TLS/SSL	26
16.	Node.js Debugger	29
17.	Node.js process	29
18.	Node.js Child process	31
19.	Node.js Buffers	34
20.	Node.js Streams	37
21.	Node.js File streams	39
22.	Node.js Path	44
23.	Node.js string.Decoder	46
24.	Node.js Query string	47
25.	Node.js ZLIB	48
26.	Node.js Assertion	49
27.	Node.js V8	50

28.	Node.js callbacks	52
29.	Node.js Events	54
30.	Node.js Punycode	56
31.	Node.js TTY	58
32.	Node.js Web Modules	61
33.	MySQL Create Connection	64
34.	MySQL Create Database	64
35.	MySQL Create Table	65
36.	MySQL Insert Record	68
37.	MySQL update Record	70
38	MySQL Delete Record	71
39.	MySQL Select Record	72
40.	MySQL Select Unique	73
41.	My SQL Drop Table	75
	• Node.js Mongo DB	
42.	Create Connection	76
43.	Create Database	76
44.	Create Collection	77
45.	MongoDB Insert	78
46.	MongoDB Select	80
47.	MongoDB Query	81
48	MongoDB Sorting	82
49.	MongoDB Remove	84

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.
Many of the basic modules of Node.js are written in JavaScript. Node.js is mostly use to run real-time server applications.
The definition given by its official documentation is as follows:
?Node.js is a platform built on chorme'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.

Node.js = Runtime Environment + JavaScript Library

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. Extermely 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 ie. 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 notifaction 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.

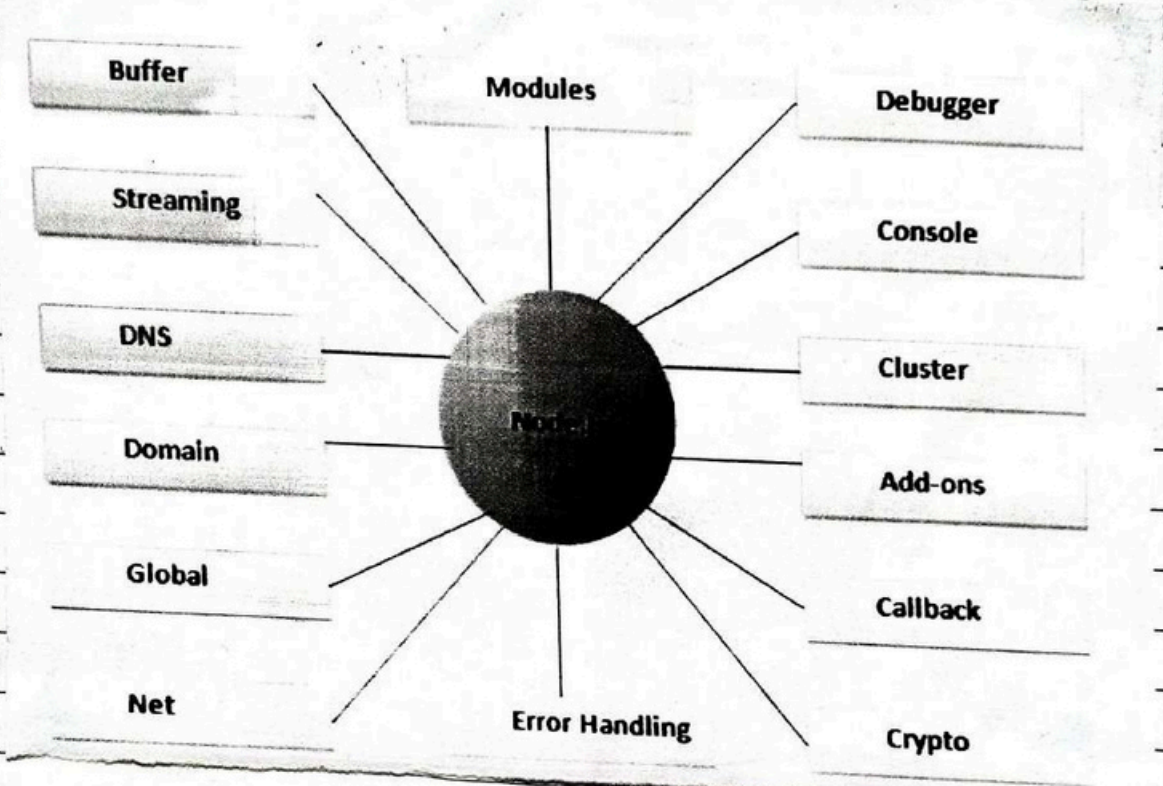
5. No buffering:
Node.js cuts down the overall processing time while uploading audio and video files. Node.js applications never buffer any data. These applications simply output the data in chunks.

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.

Different parts of Node.js

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



Install Node.js on Windows

To install and setup an environment for Node.js, you need the following two softWares available on your computer:

1. Text Editor
2. Node.js Binary installable

Text Editor:

The text editor is used to type your program. For example: Notepad is used in Windows, Vim or vi can be used on Windows as well as Linux or UNIX. The name and version of the text editor can be different

from operating system to operating system. The files created with text editor are called source files and contain program source code. The source files for Node.js programs are typically named with the extension ".js".

The Node.js Runtime:

The source code written in source file is simply Javascript. It is interpreted and executed by the Nodejs interpreter.

How to download Node.js:

You can download the latest version of Node.js installable archive file from https://nodejs.org/en/

Node.js First Example

There can be console-based and Web-based node.js applications.

Node.js console-based Example

File: console_example1.js

```
Console.log ("Hello JavaTpoint");
```

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

```
node console_example1.js
```

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.jsmodule

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:

```
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 responce parameters and Write the Sample Implementation to return "Hello World". For example:

```
http.createServer(function(request, response) {
    //Send the HTTP header
    // HTTP Status: 200: OK
    // Content Type: text/plain
    response.writeHead(200,{'Content-type': 'text/plain'});
    //Send the response body as "Hello World"
    response.end('Hello World\n');
}).listen(8081);
    // Console will print the message.
    console.log('Server running at http://127.0.0.1:8081/');
```

3. Combine step 1 and step2 together in a file named "main.js".

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 and node.js stream:

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

Node.js console.log()

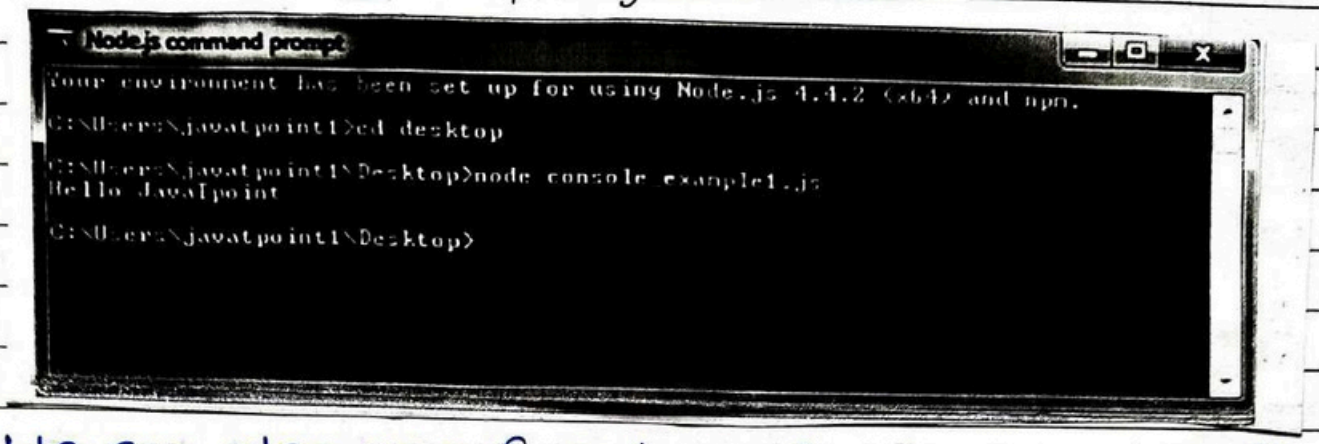
The console.log() function is used to display simple message on console.

```
Console.log('Hello JavaTpoint');
```

Open Node.js Command prompt and run the following

Code:

node console_example1.js



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

File: console_example2.js

```
Console.log('Hello %s','JavaTpoint');
```

Node.js console.error()

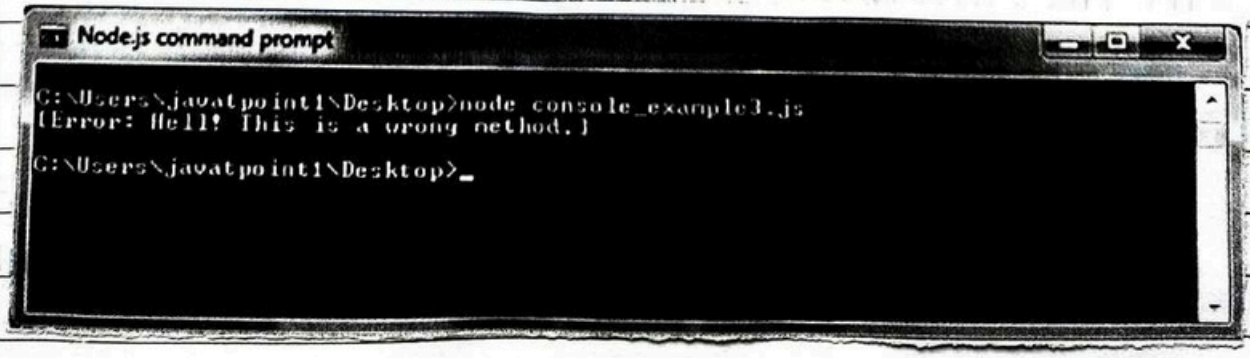
The console.error() function is used to render error message on console.

File: console_example3.js

```
Console.error(newError('Hell! This is a Wrong method.'));
```

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

node console_example3.js



Node.js Console.Warn()

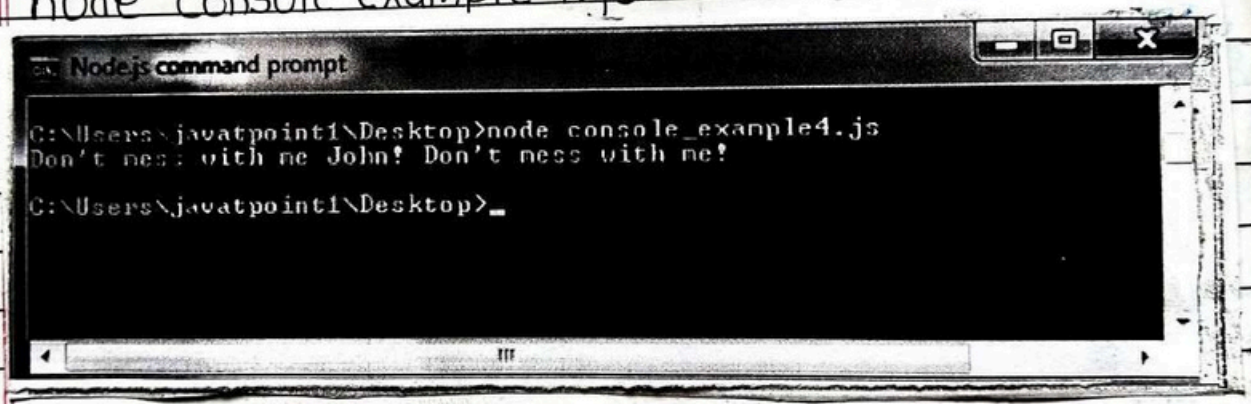
The Console.Warn() function is used to display Warning message on console.

File: Console_example4.js

```
Const name = 'John';
Console.Warn(`Don't mess with me ${name}! Don't mess with me!`);
```

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

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 Shall where you can enter the commands and the system responds with and 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 read 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.

Node.js Simple expressions

After starting REPL node command prompt put any mathematical expression:

Example : >10 + 20 -5
25

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: >a = 50
50
> var b=50
undefined
> a+b
100

Node.js Multiline expressions

Node REPL supports multiline expressions like JavaScript.

Example

```
Var x = 0
Undefined
> do {
... x++;
... console.log("x:" + x );
... } While (x<10);
```

Node.js Underscore Variable

You can also use underscore_ to get the last result.

Example

```
C:\Users\javatpoint 1\Desktop>node
> Var a = 50
Undefined
> Var b = 50
Undefined
> a+b
100
```

Node.JS REPL Commands

1. Ctrl + C:
   It is used to terminate the current command.

2. Ctrl + C twice:
   It is terminates the node repl.

3. Ctrl + d :

It termintes the node repl.

4. up/down keys:
It is used to see command history and modify previous commands.

5. tab keys:
It specifies the list of current command.

6. help:
It specifies the list of all commands.

7. break:
It is used to exit from multi-line expressions.

8. Clear:
It is used to exit from multi-line expressions.

9. .save filename:
It saves current node repl session to a file.

10. .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 version after that v0.6.3. You can check the version by opening Node.js command prompt and typing the following command:

```
npm version
```

Installing Modules using npm

```
npm install <module Name>
```

Uninstalling a Module

```
npm uninstall express
```

Searching a module

```
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.

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 set 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.

12. --trace-Warnings
It prints stack traces for process warnings.

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-chiper-list=list
It specifies an alternative default tls chiper 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 disalbed from script code)

21. --icu-data-dir=file

It specifies ICU data load path.

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.

Node.js __dirname

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

```
Console.log(__dirname)
```

Node.js __filename

It specifies the filename of the code being exected. This is the resolved asolute path of this code file. The value inside a module is the path to that module file.
File: global-example2.js

```
Console.log(__filename);
```

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

node global – example2.js

Node.js Console

click here to get details of Console class.
http://www.javatpoint.com/nodejs-console

Node.js Buffer

Click here to get details of Buffer class.
http://www.javatpoint.com/nodejs-buffers

Node.js Timer Functions

Click here to get details of Timer functions.
http://www.javatpoint.com/nodejs-timer

Node.js OS

Node.js OS provides some basic operating-system related utility functions.

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. **O.S. 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 method 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 the subset of the password file entry for the current effective user.

Node.js Timer

Node.js Timer functions are global functions. You don't need to use require() function in order to use 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

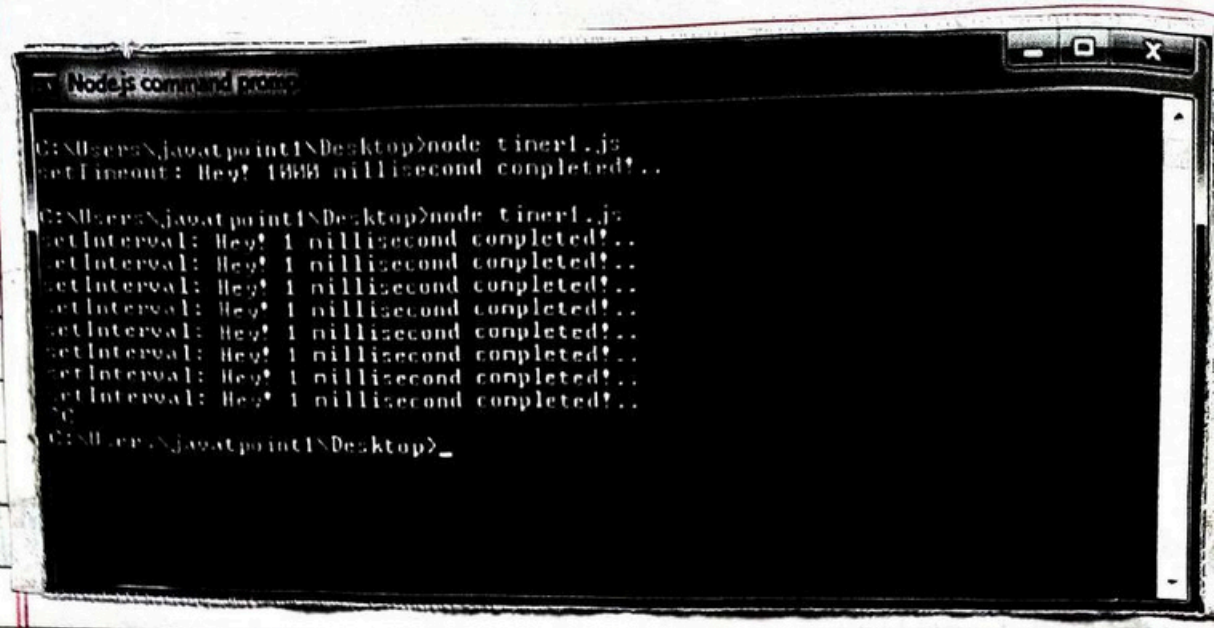
This example will set a time interval of 1000 millisecond and the specified comment will be displayed after every 1000 millisecond until you terminate.

File: timer1.js

```
setInterval(function(){
console.log("SetInterval:Hey! 1 millisecond Completed!..");
}, 1000);
```

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

```
Node timer1.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
- Use-specified errors
- Assertion errors

Node.js Errors Example

Let's take an example to deplay standard JavaScript error- Reference Error.

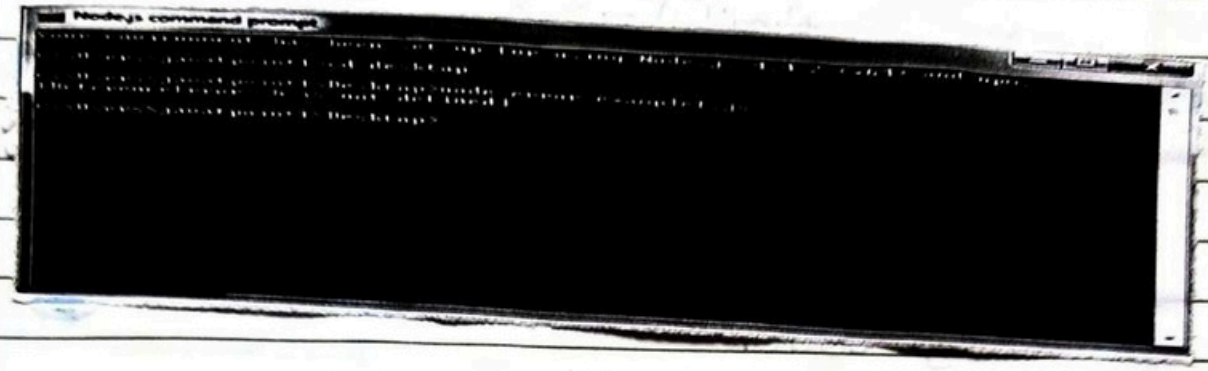
File error_example1.js

```
// Throws with a ReferenceError because b is undefined
 try {
   const a = 1;
   const c = a+b;
 } catch(err) {
   console.log(err);
 }
```

Open Node.js Command prompt and run the

following code:

node error example 1.js



Node.js DNS

The Node.js DNS module contains methods to get information of given hostname. Lets 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.resolve mx(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

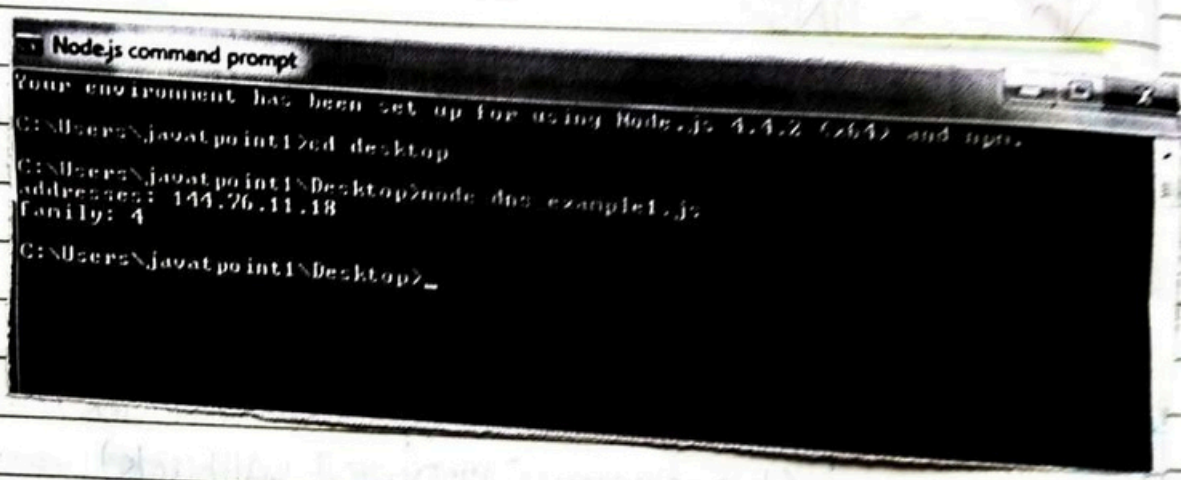
Let's see the example of dns.lookup() function.

File: dns_example1.js

```
Const dns = require('dns');
dns.lookup('www.javatpoint.com',(err, addresses,
                family) => {
  console.log('addresses:', addresses);
  console.log('family:', family);
});
```

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

```
node dns_example1.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

```
const net = require('net');
Var server = net.createServer((socket) => {
   socket.end('goodbye\n');
}).on('error', (err) => {
   // handle errors here
   throw err;
});
// grab a random port.
 server.listen(() => {
  address = server.address();
  console.log('opened server on %j', address);
});
```

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

```
node net_server.js
```

Node.js net example 1

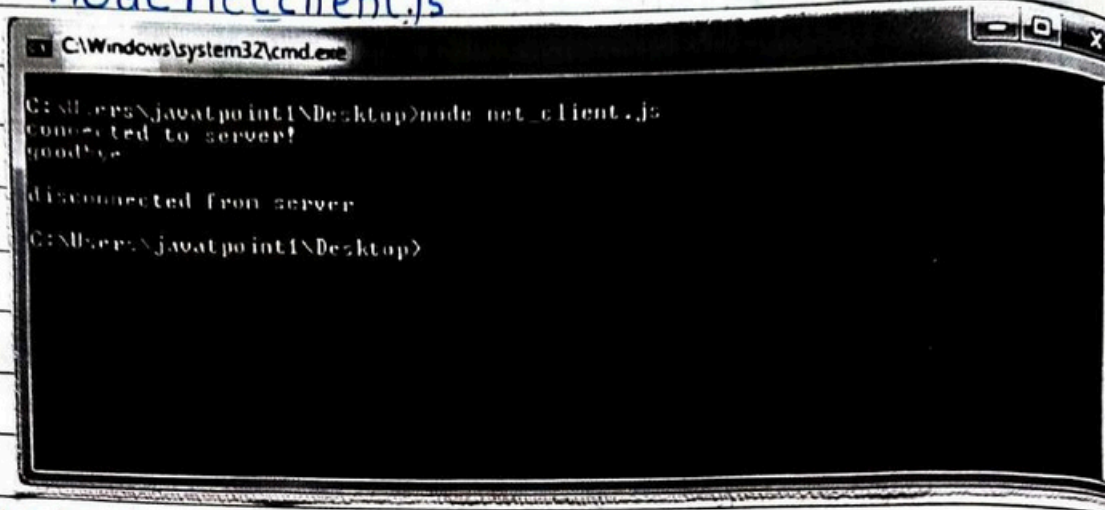
Client:

File: net_client.js

```
Const net = require('net');
Const client = net.connect({port:50302}, () => {//use same
                                Port of server
   cosnole.log('connected to server!');
   client.write('World!\r\n');
});
   client.on('data', (data) => {
   console.log(data.toString());
   client.end();
});
   Client.on('end', () => {
   console.log('disconnected from server');
});
```

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

node net_client.js



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 it 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 secretkey that results in a single final hash.

Encryption Example using Hash and HMAC

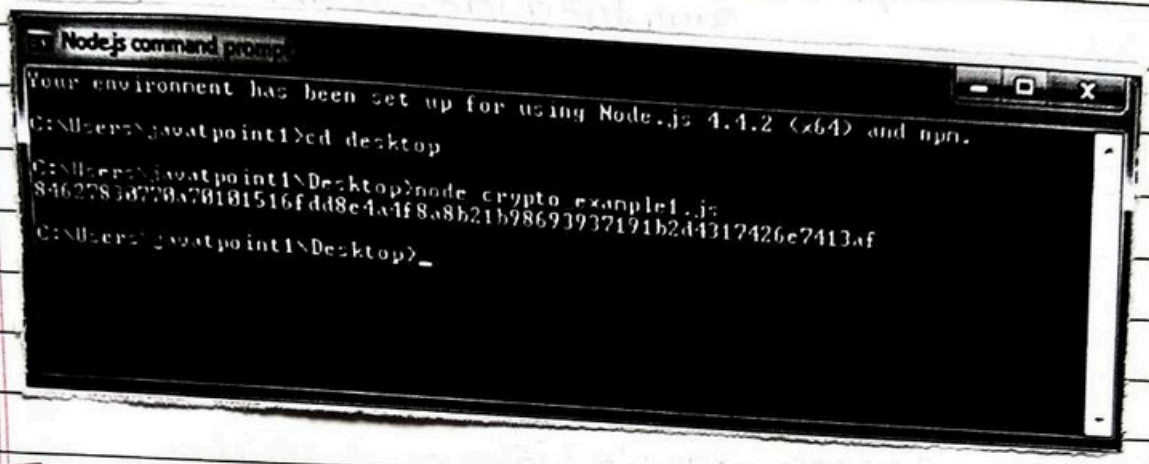
File: crypto_example1.js

```
const crypto = require('crypto');
const secret = 'abcdefg';
```

```
Const hash = crypto.createHmac ('sha256', secret)
             .update('Welcome to JavaTpoint)
             .digest('hex');
console.log(hash);
```

Open Nodejs Command prompt and run the following c

```
node crypto_example1.js
```



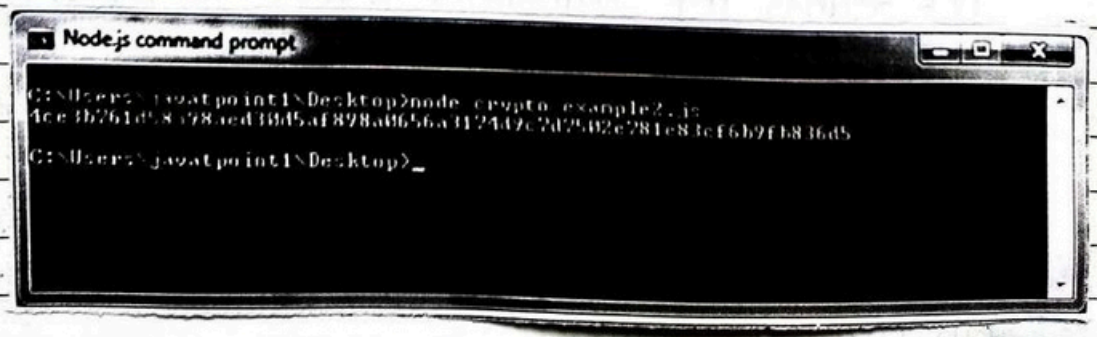
Encryption example using cipher

File: crypto_example2.js

```
Const crypto = require('crypto')
const cipher = crypto.createCipher('aes192', 'a password');
Var encrypted= Cipher.update('Hello JavaTpoint', 'utf8', 'hex'
encrypted + = cipher. final ('hex');
console.log(encrypted);
```

Open Node.js Command prompt and run the following Coo

```
node crypto_example2.js
```



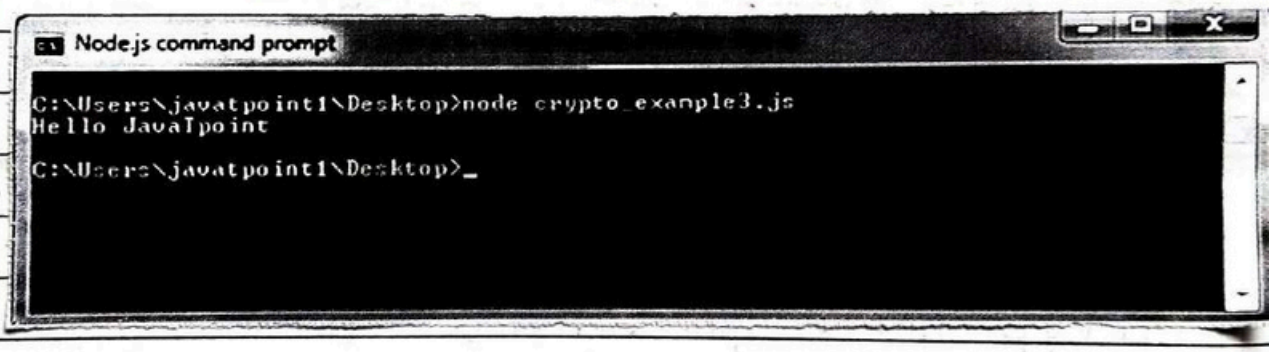
Decryption example using Decipher

File: crypto_example3.js

```
Const crypto = require ('crypto');
Const decipher = Crypto.CreateDecipher ('aes192',
                                        'a password');
Var encrypted = '4ce3b761d58398aed30d5af898906
                 3174d9c7d7502e781e83c
Var decrypted = decipher. update (encrypted, 'hex',
                                  'utf8');
decrypted + = decipher. final ('utf8');
console.log (decrypted);
```

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

```
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.

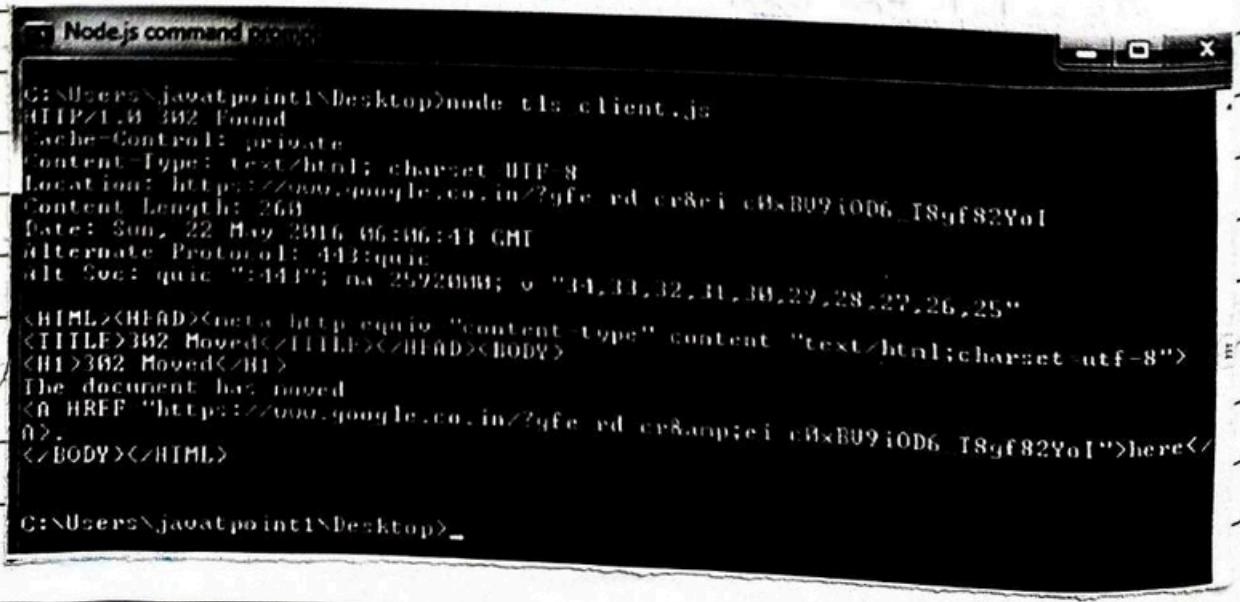
```
Var tls = require('tls');
```

Nodejs TLS Client example

File: tls_client.js

```
tls = require ('tls');
function connected(stream){
    if (stream){
      //Socket connected
    stream.Write("GET/HTTP/1.0\n\rHost:encrypted.
                  google.com:443\n\r\n\r");
  } else{
    console.log("Connection failed");
  }
}
 //needed to keep socket variable in scope
 Var dummy = this;
 //try to connect to the server
 dummy.socket = tls.connect(443,'encrypted.google.com
                            function(){
   //Callback called only after successful socket connection.
   dummy.connected = true;
   if(dummy.socket.authorized){
    //authorization successful
```

```
        dummy.socket.setEncoding('utf-8');
        connected(dummy.socket);
    }else{
        //authorization failed
        console.log(dummy.socket.authorizationError);
        connected(null)
    }
});
dummy.socket.addListener('data', function(data){
    //received data
    console.log(data);
});
dummy.socket.addListener('error', function(error){
    if(!dummy.connected){
        //socket was not connected, notify callback
        connected(null);
    }
    console.log("FAIL");
    console.log(error);
});
dummy.socket.addListener('close', function(){
    // do something
});
```



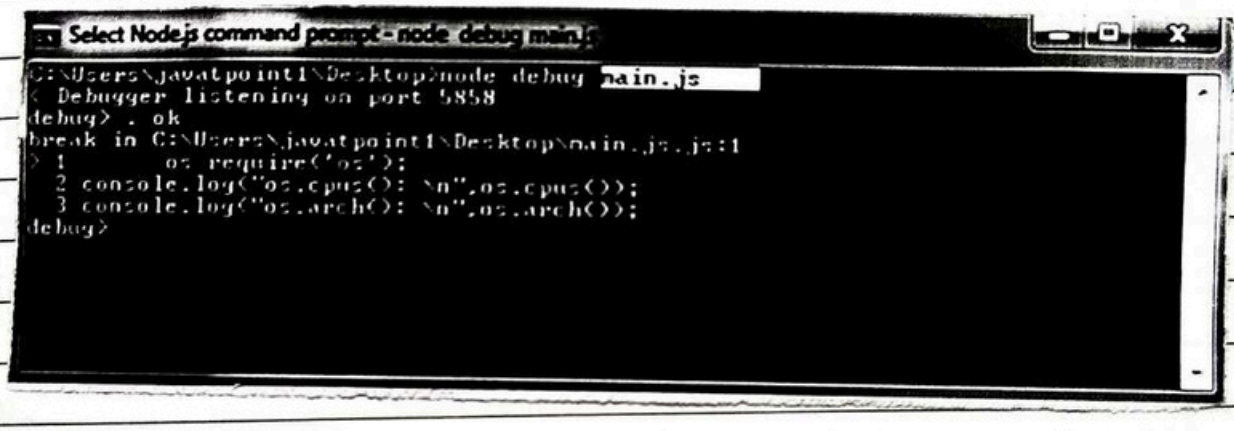
Node.js Debugger

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

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

Example

node debug main.js



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 a global object, an instance of Event-Emitter, 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: 'drawin', '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

File: process_example1.js

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

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
memory-usage()	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

File: process_example3.js

```
Console.log(`current directory: ${process.cwd()}`);
Console.log(`uptime: ${process.uptime()}`);
```

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:

Node.js child_process.exec() method

The child_process.exec() method runs a command in a console and buffers the output.

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 speifies 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

File: child_process_example1.js

```
Const exec = require('child_process');
exec('my.bat', (err, stdout, stderr) => {
  if (err) {
    console.error (err);
    return;
  }
  console.log (stdout);
});
```

Node.js child_process.spawn() method

The child_process.spawn() method launches a new process with a given command. This method returns streams and it is generally used when the process returns large amount of data.

```
child_process.spawn(command[,a[,options]])
```

Node.js child_process.spawn() example

File: support.js

```
Console.log("Child Process" + process.argv[2] + "executed.");
```

File: master.js

```
const fs = require('fs');
const child_process = require ('child_process');
for (var i =0; i<3; i++ ) {
  Var Worker Process = child_process.spawn('node,support
 WorkerProcess.stdout.on ('data', function(data){
    Console.log('stdout:' + data);
});
Worker Process.Stderr.on('data', function(data) {
    Console.log ('stderr:' + data)
});
```

```
WorkerProcess.on('close', function
    Console.log('child process exited With code'+code);
});
}
```

Node.js child_process.fork() method

This 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 childProccss instance.

```
Child_process.fork(modulePath[, a[, options]])
```

Node.js child_process.fork() example

File : Support.js

```
Const fs = require('fs')
Const Child_process = require('child_process');
 for (Var i = 0; i < 3; i++) {
 Var Worker_process = child_process.fork("Support.js", [i]
 Worker_process.on('close', function
      Console.log('Child process exited With code'+code);
    });
}
```

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 octect 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 to 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:

```
Var buf = new Buffer(10);
```

2. Create a buffer from array:
Following is the Shytax to create a Buffer from a given array:

```
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:

```
Var buf = new Buffer("Simply Easy Learning", "utf-8");
```

Node.js Writing to buffers

```
buf.Write(string[, offset][, length][, encoding])
```

Example
File: main.js

```
buf = new Buffer(256);
```

```
len = buf.write("Simply Easy Learning");
console.log ("Octets written: " + len);
```

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

```
node main.js
```

Node.js Reading from buffers

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

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

Example

File: main.js

```
buf = new Buffer(26);
for (var i = 0 ; i < 26; i++) {
  buf[i] = i + 97;
}
console.log(buf.toString('ascii')); // outputs: abcdefghijkl
                                    mnopqrstuvwxyz
console.log(buf.tostring('ascii',0,5)); //outputs: abcde
console.log(buf.toString('utf8',0,5)); //outputs: abcde
console.log(buf.tostring(undefined,0,5)); // encoding
        defaulfs to 'utf8'; outputs abcde
```

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

```
node main.js
```

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 has been flushed to underlying stream system.

Node.js Reading from stream

File: main.js

```
Var fs = require ("fs");
Var data = '';
 //Create a readable stream
Var readerStream = fs.createReadStream('input.txt');
 //Set the encoding to be utf8
 readerStream.setEncoding('UTF8');
 //Handle Stream events --> data, end, and error
 readerStream.on('data', function(chunk) {
    data += chunk;
 });
 readerStream.on('end', function() {
    console.log(data);
 });
 readerStream.on('error', function(err){
    console.log(err.stack);
 });
 console.log("Program Ended");
```

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

```
node main.js
```

Node.js Writing to stream

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

File : main.js

```
Var fs = require ("fs");
Var data = 'A solution of all Technology';
// Create a Writable stream.
Var Writer Stream = fs.createWriteStream ('output.txt')
// Write the data to stream with encoding to be ut
Writer Stream.Write(data,'UTF8');
// Mark the end of file
WriterStream.end();
// Handle Stream events --> finish, and error
WriterStream.on ('finish', function(){
    Console.log ("Write Completed.");
});
WriterStream.on('error', function(){
    Console.log(err.stack);
});
Console.log ("Program Ended");
```

Now open the Node.js Command prompt and run the main.js:

```
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:

```
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.

Lets take an example:
Create a text file named "input.txt" having the following content.
File: Input.txt
Javatpoint is a one of the best online tutorial website to learn different technologies in a very easy and efficient manner.
Lets take an example to create a JavaScript file named "main.js" having the following code:
File: main.js

```
Var fs = require("fs");
  //Asynchronous read
fs.readFile('input.txt', function(err, data) {
   if(err){
      return console.error(err)
   }
   console.log("Asynchronous read:" + data.toString());
});
// Synchronous read
Var data = fs.readFileSync('input.txt');
Console.log("Synchronous read:" + data.toString());
Console.log("Program Ended");
```

Node.js Open a file

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

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

```
Var fs = require("fs");
//Asynchronous - Opening File
Console.log("Going to open file!");
fs.open('input.txt', 'r+' function(err, fd) {
    if (err){
        return console.error(err);
    }
    console.log("File opened sucessfully!");
});
```

Node.js File Information Method

fs.Stat (path, callback)

Node.js fs. Stats Class Methods

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.

Lets take an example to create a JavaScript file named main.js having the following code:
File: main.js

```
Var fs = require ("fs");
 console.log ('Going to get file info!");
 fs.stat ('input.txt' function (err,Stats){
    if (err) {
       return console.error(err);
    }
    Console.log (stats);
    console.log ("Got file info successfully!");
    // check file type
    console.log ("isfile?" + stats.isfile());
    console.log("is Directory?" +status.isDirectory()
});
```

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:

```
Var path = require("path")
```

Node.js Path Methods

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.isabolute(path)
It determines Whether path is an absolute path. an absoulte path Will always resolve 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 is used to solve the relative it returns the directory name of a path. It is similar to the Unix base-dir-

name 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 parth 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

```
Var path = require ("path");
// Normalization
Console.log('normalization:' + normalize('/sssit/javatpoint
            //node/newfolder/tab/..'));
// Join
console.log ('joint path: ' + path.join('/sssit', 'javatpoint,
            'node/newfolder', 'tab', '..'));
// Resolve
Console.log ('resolve: ' + path.resolve('path_example.js')
//Extension
```

```
console.log('ext name: ' + path.extname('path_example.js'));
```

Node.js String.Decoder

The Node.js String.Decoder 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').StringDecoder;

Node.js StringDecoder Methods.

StringDecoder class has two methods only.

- 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.

File: stringDecoder_example1.js

```
Const StringDecoder = require('string_decoder').StringDecoder
const decoder = new StringDecoder('utf8');
const buf1 = new Buffer('this is a test');
console.log(decoder.write(buf1));//prints: this is a test
const buf2 = new Buffer('74686973206973206120746573743',
                        'hex');
console.log(decoder.write(buf2);//prints: this is a test
Const buf3 = Buffer.from([0x62,0x75,0x66,0x66,0x65,0x72]);
console.log(decoder.write(buf3));//prints: buffer
```

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.

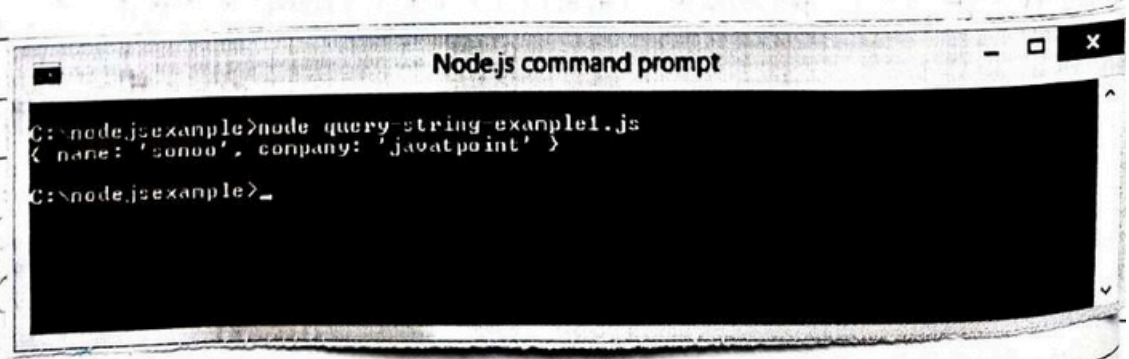
1. querystring.parse(str[,sep][,eq][,options])
Converts query string into JSON object.

2. querystring.stringify(obj[,sep][,eq][,options])
Converts JSON object into query string.

Node.js Query String Example 1: parse()

File: query_string_example-1.js

```
Querystring = require('querystring')
const obj1 = querystring.parse('name=sonoo&
                company=javatpoint');
console.log(obj1);
```



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:

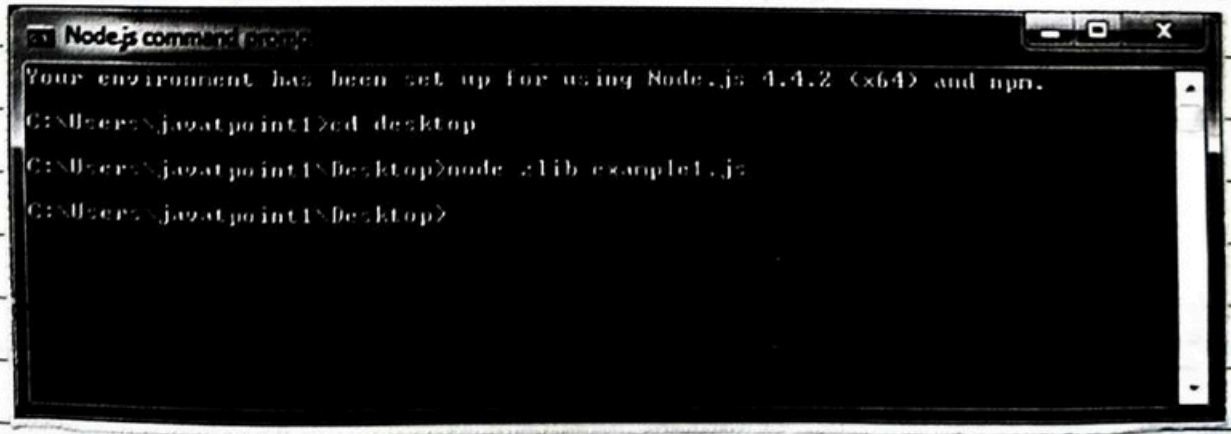
```
Const zlib = require('zlib');
```

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

File: Zlib_example1.js

```
Const zlib = require('zlib');
Constgzip = zlib.createGzip();
Const fs = require('fs');
Const inp = fs.createReadStream('input.txt');
Const out = fs.createWritestream('input.txt.gz');
inp.pipe(gzip).pipe(out)
```



Node.js ZLIB Example : Decompress File

File: Zlib_example2.js

```
Const zlib = require('zlib');
Const unzip = zlib.createUnzip();
```

```
Const fs = require('fs');
Const inp = fs.createReadStream('input.txt.gz');
Const out = fs.createWritestream('input2.txt');
inp.pipe(unzip).pipe(out);
```

Node.js Assertion Testing

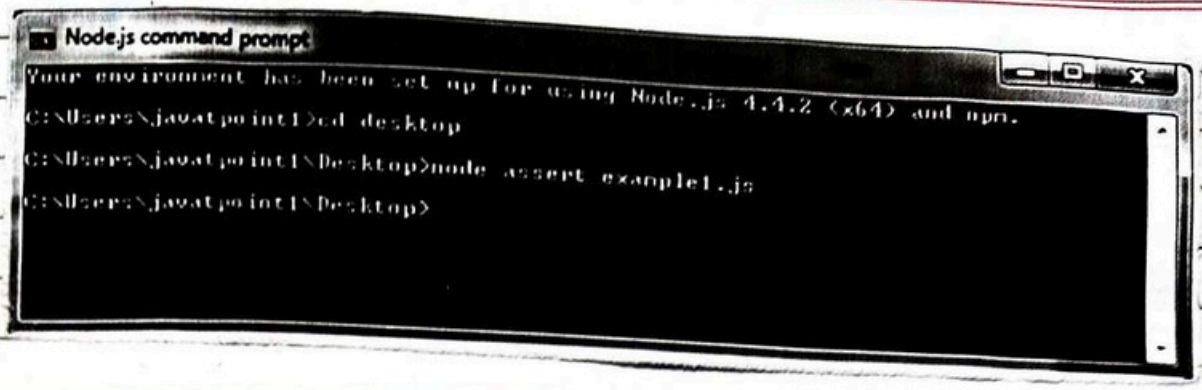
The Node.js Assert is the most elementary way to write tests. It provides no feedback when running your test unless one fails. The assert module provides a simple set of assertion tests that can be used to test invariants. The module is intended for internal use by Node.js, but can be used in application code via require('assert'). However assert is not a testing framework and cannot be used as general purpose assertion library.

Node.js Assert Example

File: assert_example1.js

```
Var assert = require('assert');
function add(a,b){
   return a+b;
}
Varexpected = add(1,2);
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.



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').

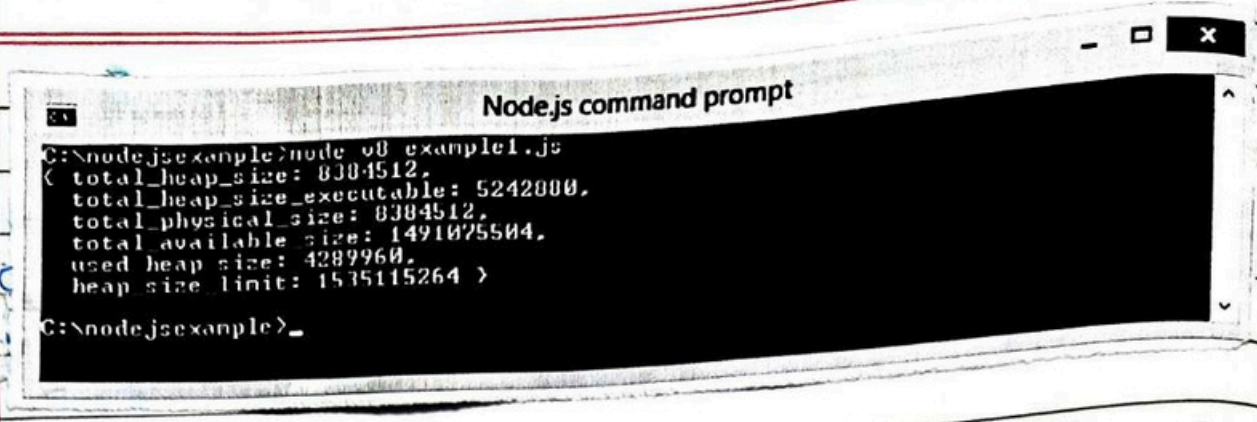
```
Const V8 = require('V8');
```

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.

File: V8_example1.js

```
Const V8 = require('V8');
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

```
Const V8 = require('V8');
Console.log (V8.getHeapSpaceStatistics());
```

```
C:\nodejsexample>node v8-example2.js
[ { space_name: 'new_space',
    space_size: 2097152,
    space_used_size: 1021376,
    space_available_size: 10560,
    physical_space_size: 2097152 },
  { space_name: 'old_space',
    space_size: 3063808,
    space_used_size: 2465304,
    space_available_size: 0,
    physical_space_size: 3063808 },
  { space_name: 'code_space',
    space_size: 2097152,
    space_used_size: 613152,
    space_available_size: 2304,
    physical_space_size: 2097152 },
  { space_name: 'map_space',
    space_size: 1126400,
    space_used_size: 171488,
    space_available_size: 0,
    physical_space_size: 1126400 },
  { space_name: 'large_object_space',
    space_size: 0,
    space_used_size: 0,
    space_available_size: 1491062528,
    physical_space_size: 0 } ]
C:\nodejsexample>
```

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.

Blocking Code Example

Follow these steps:

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

   Javatpoint is an online platform providing self learning tutorials on different technologies, in a very simple language.

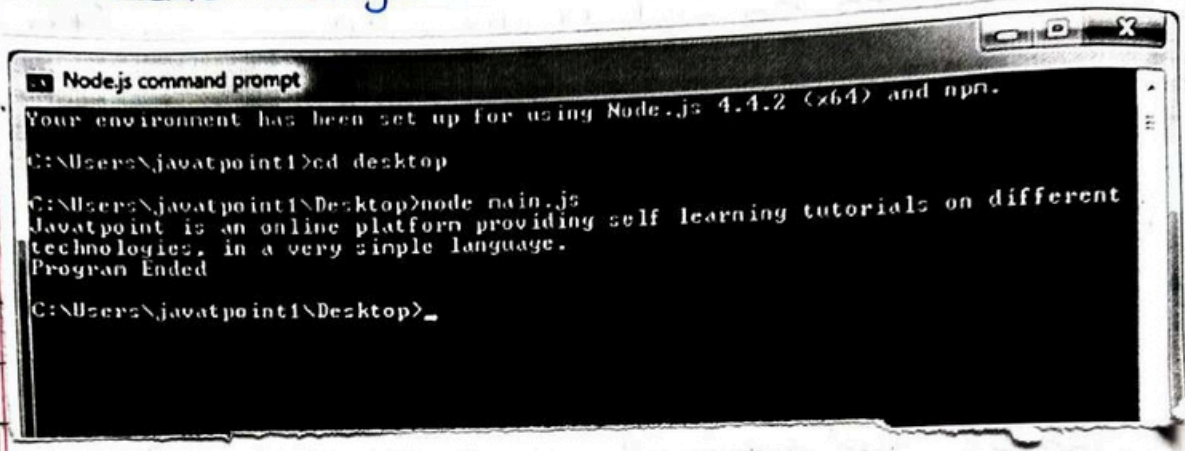
2. Create a JavaScript file named main.js having the following code:

```
Var fs = require("fs");
Var data = fs.readFileSync('input.txt');
```

```
Console.log (data. toString());
Console.log ("program Ended");
```

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

```
Ndde main.js
```



Non Blocking Code Example

Follow these steps:

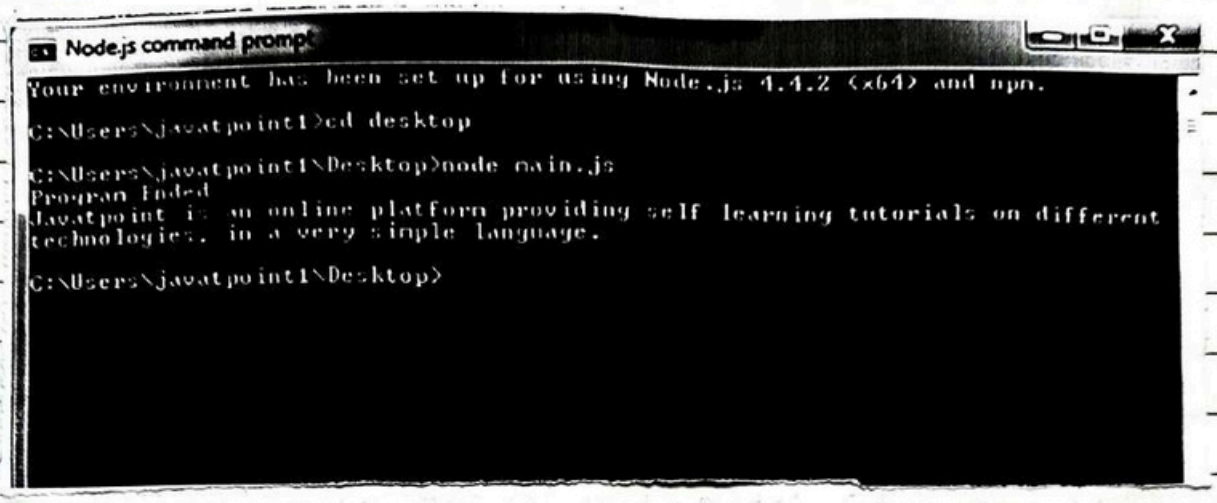
1. Create a text file named input.txt having the following content:
   Javatpoint is an online platform providing self learning tutorials on different technologies, in a very simple language.

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

```
Var fs = require ("fs");
fs. readFile('input.txt', function (err, data) {
    If(err) return console.error(err)
    Console. log (data. tostring());
});
Console.log ("Program Ended");
```

3. Open the Node.js Command prompt and execute the following Code.

node main.js

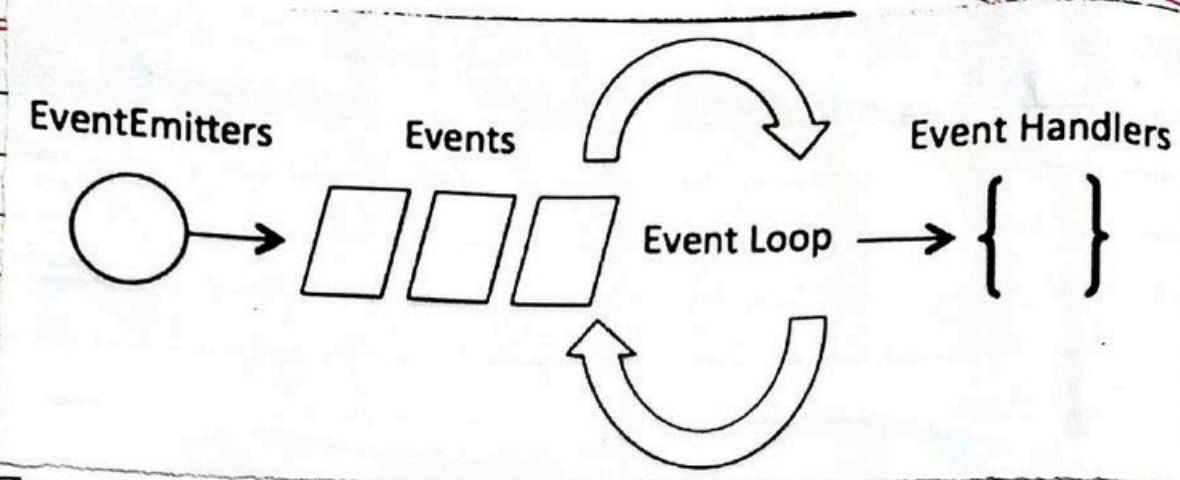


Node.js Events

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 observe 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 intiates its variables, declares function and then simply waits for event to occur. It is the one of the reson why Node.js is preety 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.



EventEmitter class to bind event and event listener:

```
//Import events module
Var events = require('events');
//Create an eventEmitter object
Var eventEmitter = new events.EventEmitter();
```

To bind event handler with an event:

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

To fire an event:

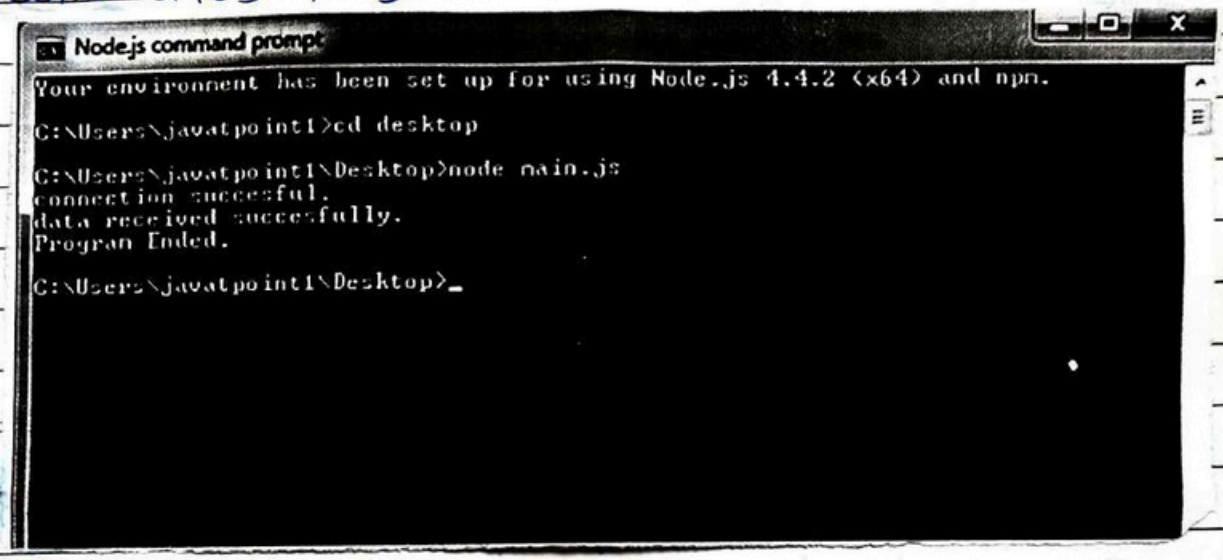
```
// Fire an event
eventEmitter.emit('eventName');
```

Node.js Event Example

File: main.js

```
//Import events module
Var events = require('events');
// Create an eventEmitter object
Var eventEmitter = new events.EventEmitter();
//Create an event handler as follows
Var connectHandler = function connected() {
    console.log('connection succesful.');
// Fire the data_received event
  eventEmitter.emit('data_received');
}
```

```
// Bind the Connection event With the handler.
eventEmitter.on('connection', connectHandler);
// Bind the data received event with the anonymous funct
eventEmitter.on('data received', function() {
    Console.log('data received succesfully.');
});
// Fire the connection event
eventEmitter.emit('connection');
Console.log("program Ended.");
```



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)

Punycode in Node.js

Punycode.js is bundled with Node.js v0.6.2 and later version. 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.

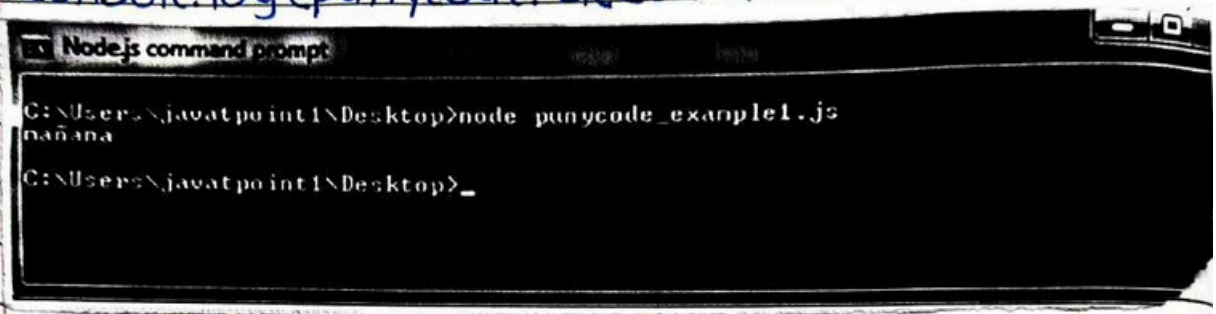
punycode = require('punycode');

punycode.decode(string)

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

File: punycode_example1.js

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

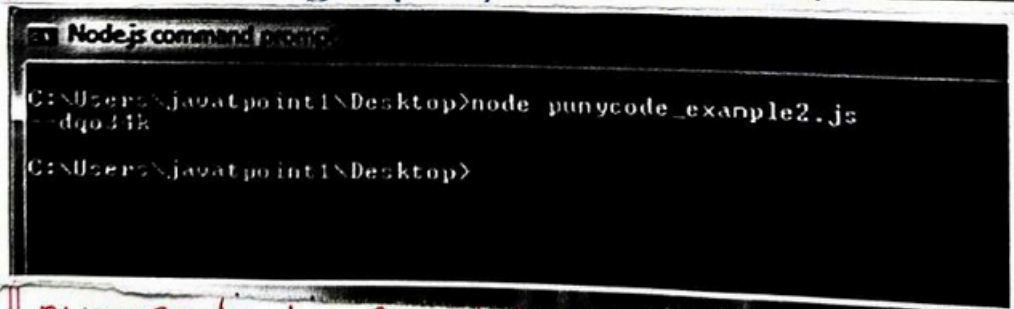


punycode.encode(string)

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

File: punycode_example2.js

```
punycode = require('punycode');
console.log(punycode.encode('
```

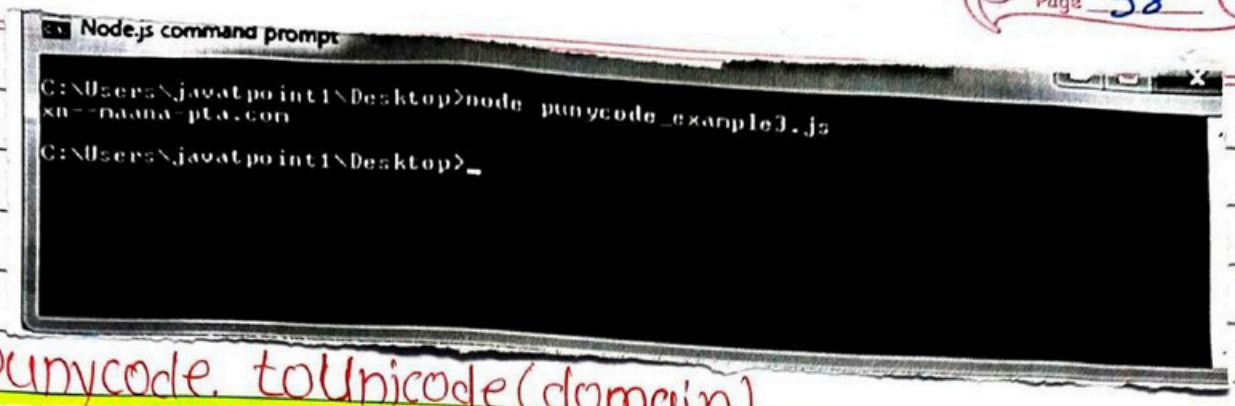


punycode.toASCII(domain)

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

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

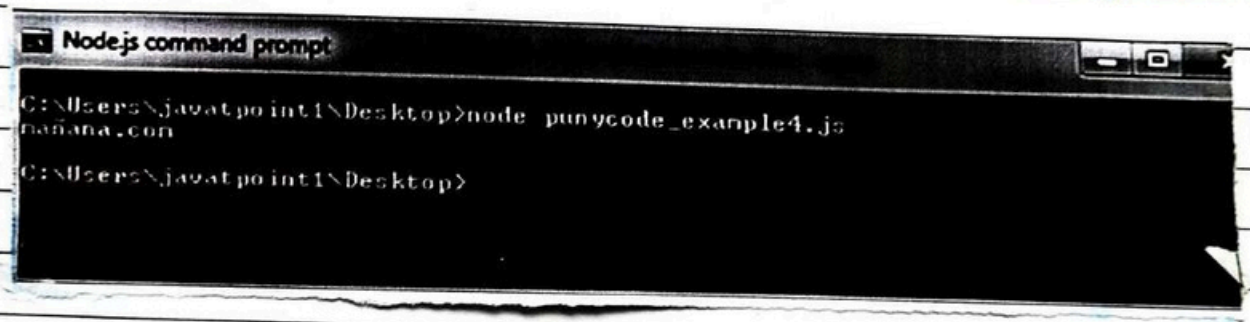


punycode.toUnicode(domain)

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

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



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.

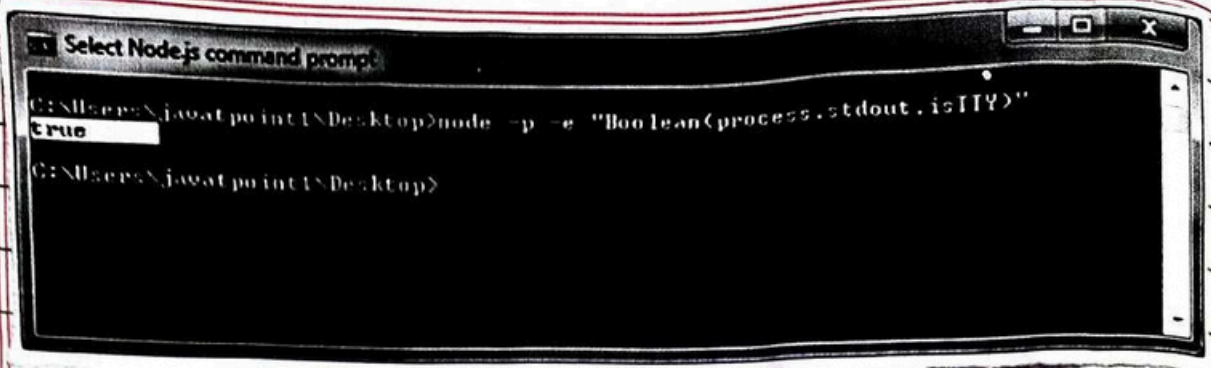
you have to use require('tty') to access this module

```
Var tty = require('tty')
```

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

- 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:



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 it initialized to false. It specifies the current "raw" state of the tty.Readstream Instance.

rs.setRawMode(mode)

It is should be true or false. It is used to set the properties of the tty.ReadStream to act either as a raw device or defalut, is Raw 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 it any Node.js program (only when isatty(1) is true).

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

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

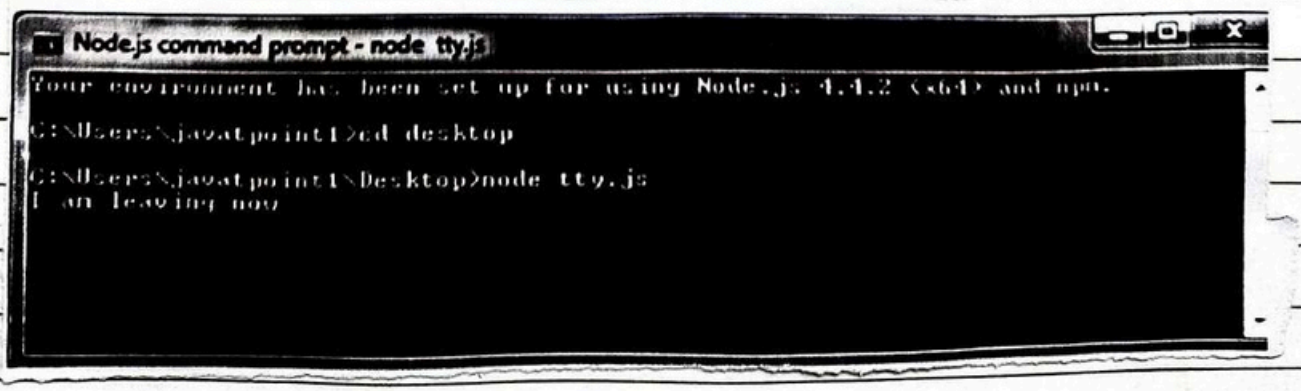
ws.columns:
It is used to give the number of columns the TTY currently has. This property gets updated on 'resize' events.

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

```
var tty = require('tty');
process.stdin.setRawMode(true);
process.stdin.resume();
 console.log('I am leaving now');
process.stdin.on('keypress', function (char, key) {
  if (key && key.ctrl && key.name == 'c') {
    process.exit()
  }
});
```



Node.js Web Module

What is Web Server

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

Web Application Architecture

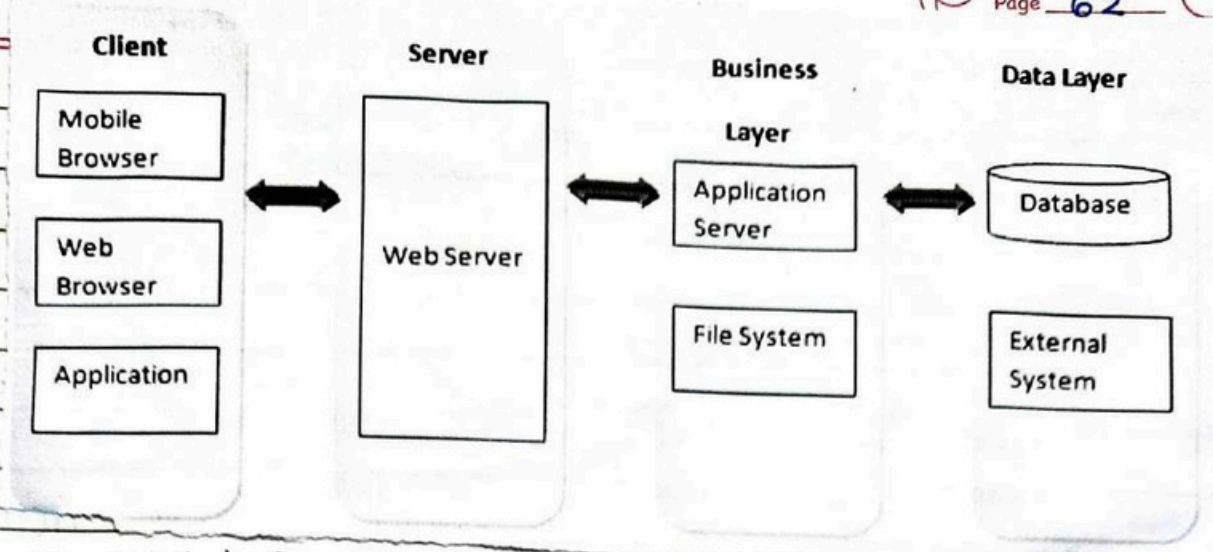
A Web application can be devided in 4 layers:

- **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 inter cepts 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 database or any sourc of data.



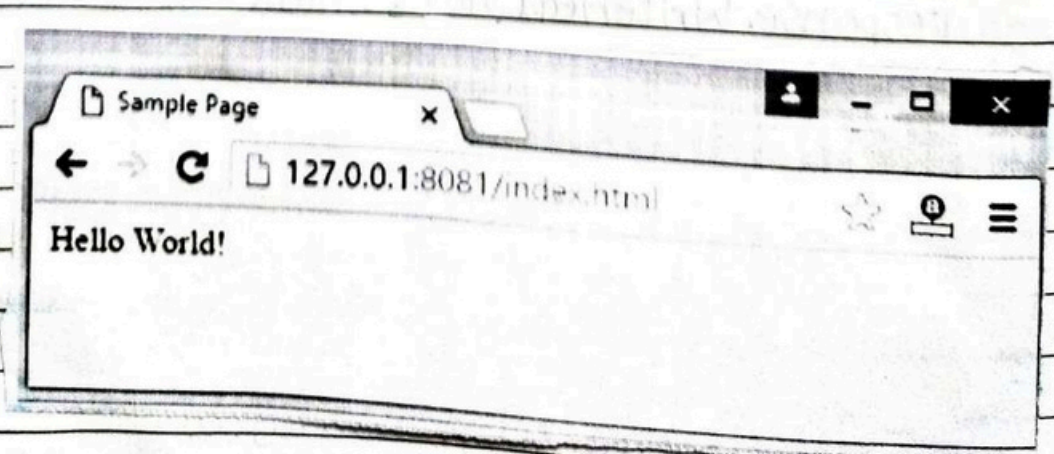
Creating Web Server Using Node.js

```
Var http = require ('http');
Var fs   = require ('fs');
Varurl   = require ('url');
// Create a Server
http.createServer(function(request,response){
   // Parse the request containing file name
   Var parthname = Url.parse(request.url).pathname;
   // Print the name of the file for which request is mode.
   Console.log("Request for" + pathname + "received.");
   // Read the requested file content from file system.
   fs.readFile(pathname.substr(1), function(err, data){
    if (err){
      Console.log(err);
      // HTTP Status: 404: NOT FOUND
      // Content Type : text/plain
      response.WriteHead(404, {'Content-Type': 'text/html'});
    }else{
      // Page found
      // HTTP Status : 200: Ok
      // Content Type: text/plain
      response.WriteHead(200, {'Content-Type': 'text/html'});
      // Write the content of the file to response body
      response.Write(data.toString());
}
```

```
    // Send the response body
    response.end();
  });
});.listen(8081);
// Console will print the message.
console.log("Server running at http://127.0.0.1:8081/");
```

Next, create an html file named index-html hav the following in the same directory where you created server.js

```
<html>
<head>
<title> Sample page </title>
</head>
<body>
Hello World!
</body>
</html>
```



Node.Js Create Connection With MySQL

Create Connection

Create a folder named "DBexample". In that folder Create a js file named "connection.js" having the following code:

```
Var mysql = require('mysql');
Var con = mysql.createConnection({
  host: "localhost",
  User: "root",
  password: "12345"
});
con.connect(function(err){
  if(err) throw err;
  Console.log("Connected!");
});
```

Node.js MySQL Create Database

CREATE DATABASE statement is used to Create a database in MySQL.

Example

For creating a database named "javatpoint":

Create a js file named Javatpoint.js having the following data in DBexample folder.

```
Var mysql = require('my sql');
Var con = mysql.createConnection({
 host: "localhost",
```

```
user: "root",
password: "12345"
});
con.connect(function(err) {
if(err) throw err;
Console.log("Connected!");
con.query("CREATE DATABASE javatpoint", function(err,
                                    result) {
if (err) throw err;
Console.log("Connected!");
con.query log("Database created");
});
});
```

```
root@aspire-pc: /home/user/Desktop/MongoDatabase
root@aspire-pc:/home/user# cd Desktop
root@aspire-pc:/home/user/Desktop# cd MongoDatabase
root@aspire-pc:/home/user/Desktop/MongoDatabase# node createdatabase.js
Database created!
root@aspire-pc:/home/user/Desktop/MongoDatabase#
```

NOde.js MySQL Create Table

CREATE TABLE command is used to create a table in MySQL. You must make it sure that you define the name of the database When you create the connection.

Example

For creating a table named "employees". Create a js file named employee.js having the following data in DBexample folder.

```
Var mysql = require('mysql');
Var con = mysql.create Connection({
host: "localhost",
```

```
user: "root",
password: "12345"
data base: "javatpoint"
});
con.connect(function(err){
if(err) throw err;
console.log("Connected!");
Var sql ="CREATE TABLE employees(id INT, name VARCHAR(255),
          age INT(3), city VARCHAR(255))";
con.query(sql, function(err, result){
 if(err) throw err;
 console.log("Table created");
 });
 });
```

```
root@aspire-pc: /home/user/Desktop/DBexample
root@aspire-pc:/home/user/Desktop# cd DBexample
root@aspire-pc:/home/user/Desktop/DBexample# node employees.js
Connected!
Table created
```

Create Table Having a Primary key

Create Primary key in new table:
Let's create a new table named "employee2" having id as primary key.
Create a js file named employee2.js having the following data in DBexample folder.

```
Var mysql = require('mysql');
Var con = mysql.createconnection({
host: "localhost",
user: "root",
password: "12345",
database: "javatpoint"
});
```

```
con.connect(function(err){
if(err) throw err;
Console.log("connected!");
Var sql = "CREATE TABLE employee2 (id INT PRIMARY
          KEY, name VARCHAR(255), age INT(3), city VARC
Con.query(sql, function(err, result){
 if(err) throw err;
 console.log("Table created");
});
});
```

```
root@aspire-pc: /home/user/Desktop/DBexample
root@aspire-pc:/home/user/Desktop/DBexample# node employee2.js
Connected!
Table created
```

Add columns in existing Table:

ALTER TABLE statement is used to add a column in an existing table. Take the already created table "employee2" and use a new column Salary. Replace the data of the "employee2" table with the following data:

```
 Var mysql = require("mysql");
 Var con = mysql.createConnection({
  host: "localhost";
  user: "root"
   password: "12345",
   database: "Javatpoint"
  });
   Con.connect(function(err){
   if(err) throw err;
   console.log("Connected!");
```

```
Var sql = "ALTER TABLE employee2 ADD COLUMN salary INT(10)";
con.query(sql, function (err, result) {
if (err) throw err;
console.log ("Table alerted");
});
});
```

```
root@aspire-pc: /home/user/Desktop/DBexample
root@aspire-pc:/home/user/Desktop/DBexample# node employee2.js
Connected!
Table altered
```

Nodejs My SQL Insert Records

INSERT INTO Statement is used to insert records in MySQL.

Example

Insert Single Record:

Insert records in "employees" table.

Create a js file named "insert" in D.B example folder and put the following data into it:

```
Var mysql = require('mysql');
Var con = mysql.createConnection({
host: "localhost",
user: "root",
password: "12345"
database: "javatpoint"
});
con.connect(function(err){
If (err) throw err;
console.log ("Connected!");
Var sql = "INSERT INTO employees (id, name, age, city)
          VALUES ('1', 'Ajeet kumar', '27', 'Allahabad')";
```

```
con.query (sql, function (err, result) {
if (err) throw err;
Console.log ("1 record inserted");
});
```

Now open command terminal and run the following command:

```
Node insert.js
```

```
root@aspire-pc: /home/user/Desktop/MongoDatabase
oot@aspire-pc:/home/user/Desktop/MongoDatabase# node insert.js
 record inserted
oot@aspire-pc:/home/user/Desktop/MongoDatabase#
```

Check the inserted record by using SELECT query:

```
SELECT * FROM employees;
```

```
root@aspire-pc: /home/user/Desktop/MongoDatabase
root@aspire-pc:/home/user/Desktop/MongoDatabase# node insertall.js
Number of records inserted: 4
root@aspire-pc:/home/user/Desktop/MongoDatabase#
```

The Result Object

When executing the insert() method a result object is returned. The result object contains information about the insertion.
It is looked like this:

```
root@aspire-pc: /home/user/Desktop/DBexample
^C
root@aspire-pc:/home/user/Desktop/DBexample# node select.js
[ RowDataPacket { id: 1, name: 'Ajeet Kumar', age: 27, city: 'Allahabad' },
  RowDataPacket { id: 2, name: 'Bharat Kumar', age: 25, city: 'Mumbai' },
  RowDataPacket { id: 3, name: 'John Cena', age: 35, city: 'Las Vegas' },
  RowDataPacket { id: 4, name: 'Ryan Cook', age: 15, city: 'CA' } ]
```

Node.js MySQL Update Records

The UPDATE command is used to update records in the table.

Example

Update city in "employees" table where id is 1.

Create a file named "update" in DBexample folder and put the following data into it.

```
Var mysql = require ('mysql');
Var con = mysql.createConnection({
host: "localhost",
user: "root"
password: "12345"
database: "javatpoint"
});
con.connect(function(err){
if (err) throw err;
Var sql = "UPDATE employees SET city = 'Delhi' WHERE
          city = 'Allahabad'";
con.query (sql, function (err, result) {
if (err) throw err;
console.log(result.affectedRows + "record (s)updated")
});
});
```

Now open command terminal and run the following command:

```
Node update.js
```

It will change the city of the id 1 is to Delhi which is prior Allahabad.

```
root@aspire-pc: /home/user/Desktop/DBexample
^C
root@aspire-pc:/home/user/Desktop/DBexample# node update.js
1 record(s) updated
```

Node.js My SQL Delete Records

The DELETE FROM command is used to delete records from the table.

Example

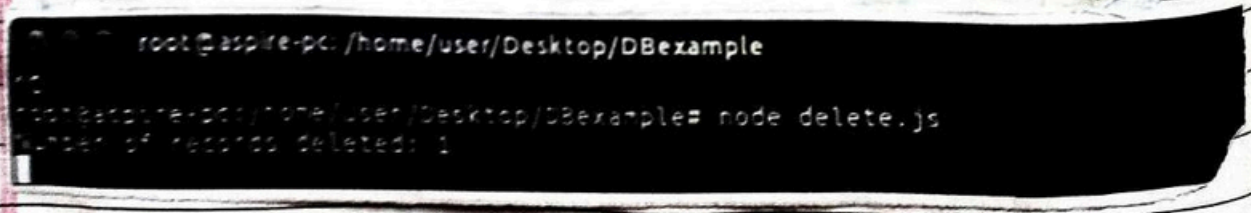
Delete employee from the table employees where city is Delhi.

Create a js file named "delete" in DBexample folder and put the following data into it:

```
Var my sql = require ('mysql');
 Var con = mysql.createconnection({
 host:"localhost"
 user: "root",
 password : "12345",
 database: "javatpoint"
 });
 con.connect(function(err){
 if (err) throw err;
 Var sql = "DELETE FROM employees WHERE city 'Delhi";
 con.query(sql, function (err, result){
 if(err) throw err;
 console.log("Number of records deleted:" + result.
                affectedRows);
 });
 });
```

Now open command terminal and run the following command:

```
Node delete.js
```



Node.js My SQL Select Records

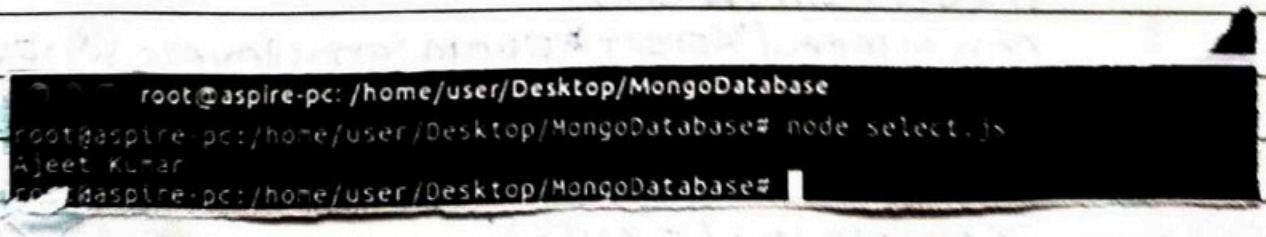
Example

Retrieve all data from the table "employees".
Create a js file named select.js having the following data in DBexample folder.

```
Var mysql = require ('mysql');
Varcon = mysql.createConnection ({
 host: "localhost",
 user: "root",
 password: "12345",
 database: "javatpoint"
 });
 con.connect(function(err){
 if(err) throw err;
 con.query("SELECT * FROM employees", function(err,result){
 if(err) throwerr:
 Console.log(result);
 });
 });
```

Now open command terminal and run the following command.

```
Node select.js
```



You can also use the statement:

```
SELECT * FROM employees;
```

```
root@aspire-pc: /home/user/Desktop/MongoDatabase
root@aspire-pc:/home/user/Desktop/MongoDatabase# node selectall.js
[ { _id: 59104e062f60cd3366ceb7da,
    name: 'Ajeet Kumar',
    age: '28',
    address: 'Delhi' },
  { _id: 59104e062f60cd3366ceb7db,
    name: 'Mahesh Sharma',
    age: '25',
    address: 'Ghaziabad' },
  { _id: 59104e062f60cd3366ceb7dc,
    name: 'Tom Moody',
    age: '31',
    address: 'CA' },
  { _id: 59104e062f60cd3366ceb7dd,
    name: 'Zahira Wasim',
    age: '19',
    address: 'Islamabad' },
  { _id: 59104e062f60cd3366ceb7de,
    name: 'Jack Ross',
    age: '45',
    address: 'London' } ]
root@aspire-pc:/home/user/Desktop/MongoDatabase#
```

Node.js MySQL SELECT Unique Record

(WHERE Clause)

Retrieve a unique data from the table "employe
Create a js file named selectwhere.js having the following data in DBexample folder.

```
Var mysql = require ('mysql');
Var con = mysql.createConnection({
host: "localhost",
user : "root",
password: "12345"
database: "javatpoint"
});
con.connect(function(err) {
if (err) throw err;
con.querry("SELECT * FROM employess WHERE id =
        '1', function(err, result) {
If (err) throw err;
 console.log (result);
});
});
```

Now open command terminal and run the

following command:

Node selectWhere.js

Node.js My SQL Select Wildcard

Retrieve a unique data by using Wildcard from the table "employees".

Create a .js file named selectwildcard.js having the following data in DBexample folder.

```
Var mysql = require('mysql');
Varcon = mysql.createConnection({
host: "localhost",
user: "root"
password: "12345"
database: "javatpoint"
});
con.connect(function(err) {
if (err) throw err.
con.query("SELECT * FROM employees WHERE city LIKE
          'A%'", function (err, result) {
if (err) throw err;
console.log (result);
});
});
```

Now open command terminal and run the following command

Node selectWildcard.js

It will retrieve the record where City start with A.

```
root@aspire-pc: /home/user/Desktop/DBexample
^C
root@aspire-pc:/home/user/Desktop/DBexample# node selectwildcard.js
[ RowDataPacket { id: 1, name: 'Ajeet Kumar', age: 27, city: 'Allahabad' } ]
```

Node.js MySQL Drop Table

The DROP TABLE command is used to delete or drop a table.

Lets drop a table named employee2.

Create a js file named "delete" in DBexample folder and put the following data into it:

```
Var mysql = require('mysql');
Var con = mysql.createConnection({
host: "localhost",
user: "root"
password: "12345",
database: "javatpoint"
});
Con.connect(function(err){
if(err) throw err;
Var sql = "DROP TABLE employee2";
Con.query(sql, function(err, result){
if(err) throw err;
Console.log("Table deleted");
});
});
```

Now open command terminal and run the following command:

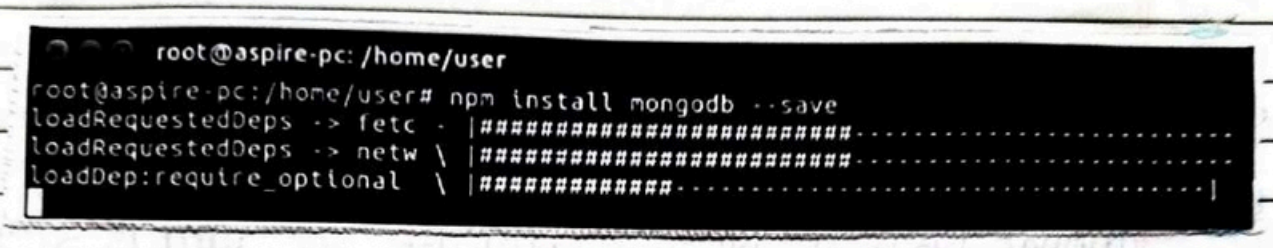
```
Node drop.js
```

```
root@aspire-pc: /home/user/Desktop/DBexample
^C
root@aspire-pc:/home/user/Desktop/DBexample# node drop.js
Table deleted
```

Node.js Create Connection With Mongo DB

MongoDb is a NOSQL database. It can be used with Node.js as a database to insert and retrieve data. Use the following command to start MongoDB Services:

```
Service mongodb start
```



Now, connection is created for further operations.

Node.js Mongo DB Create Database

To create a database in MongoDB. First create a Mongo Client and specify a connection URL with the correct ip address and the name of the database which you want to create.

Example

Create a folder named "MongoDatabase" as a database. Suppose you create it on Desktop. Create a file named "Createdatabase.js" within that folder and having the following code:

```
Var MongoClient = require('mongodb').MongoClient;
Var url = "mongodb://localhost:27017/MongoDatabase";
MongoClient.connect(url, function(err, db) {
if (err) throw err;
console.log("Database created!");
db.close();
});
```

Now open the command terminal and set the path where MongoDatabase exists. Now execute the following command:

Node create data base.js

```
root@aspire-pc: /home/user/Desktop/MongoDatabase
root@aspire-pc:/home/user# cd Desktop
root@aspire-pc:/home/user/Desktop# cd MongoDatabase
root@aspire-pc:/home/user/Desktop/MongoDatabase# node createdatabase.js
Database created!
root@aspire-pc:/home/user/Desktop/MongoDatabase#
```

Now database is created.

Node.js MongoDB Create Collection

MongoDB is a NoSQL database so data is stored in collection instead of table. Create Collection method is used to create a collection in MongoDB.

Example

Create a collection named "employees".

Create a js file named "employees.js", having the following data:

```
Var Mongo client = require('mongodb').MongoClient;
Var url = "mongodb://localhost:27017/Mongo Database";
Mongo Client.connect (url, function(err, db) {
if(err) throw err;
db.create Collection("employees", function(err, res) {
if(err) throwerr;
console.log("Collection is created!");
db.close();
});
});
```

Open the Command terminal and run the following command:

Node employees.js

```
root@aspire-pc: /home/user/Desktop/MongoDatabase
root@aspire-pc:/home/user/Desktop/MongoDatabase# node employees.js
Collection is created!
root@aspire-pc:/home/user/Desktop/MongoDatabase#
```

Now the collection is created.

Node.js MongoDB Insert Record

The insertOne method is used to insert record in MongoDB's collection. The first argument of the insertOne method is an object which contains the name and value of each field in the record you want to insert.

Example

(Insert Single record)

Insert a record in "employees" collection.

Create a js file named "insert.js", having the following code:

```
Var MongoClient = require('mongodb').MongoClient;
Var url = "mongodb://localhost:27017/MongoDatabase";
MongoClient.connect(url, function(err,db){
if (err) throw err;
Var myobj={name:"Ajeet kumar", age:"28", address: "Delhi"};
db.collection("employees").insertone(myobj, function(err,res){
if(err) throw err;
console.log("1 record inserted");
db.close();
});
});
```

Open the command terminal and run the following command:

```
Node insert.js
```

```
root@aspire-pc: /home/user/Desktop/MongoDatabase
root@aspire-pc:/home/user/Desktop/MongoDatabase# node insert.js
1 record inserted
root@aspire-pc:/home/user/Desktop/MongoDatabase#
```

Now a record is inserted in the Collection.

Insert Multiple Records

You can insert multiple records in a collection by using insert()method. The insert()method uses array of objects which contain the data you want to insert.

Example

Insert multiple records in the collection named "employees".

Create a js file name insertall.js having the following Code:

```
Var MongoClient = require('mongodb').MongoClient;
Var url = "mongodb://localhost:27017/MongoDatabase";
MongoClient.connect(url, function(err, db) {
if (err) throw err;
Var myobj = [
{name:"Mahesh sharma", age:"25", address:"Ghaziabad"},
{name:"Tom Moody", age:"31", address: "CA"},
{name: "Zahira Wasim", age:"19", address:"Islamabad"}
];
db.collection("customers").insert(myobj, function(err, res) {
if (err) throw err;
console.log("Number of records inserted: " + res.insertedCount);
db.close();
});
});
```

Open the command terminal and run the following command:

Node insertall.js

```
root@aspire-pc: /home/user/Desktop/MongoDatabase
root@aspire-pc:/home/user/Desktop/MongoDatabase# node insertall.js
Number of records inserted: 4
root@aspire-pc:/home/user/Desktop/MongoDatabase#
```

You can see here 4 records are inserted.

Node.js MongoDB Select Record

The findOne() method is used to select a single data from a collection in MongoDB. This method returns the first record of the collection.

Example

(Select Single Record)

Select the first record from the "employees" collection.

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

```
Var http = require('http');
Var MongoClient = require('mongodb').MongoClient;
Var url = "mongodb://localhost:27017/MongoDatabase";
MongoClient.connect(url, function(err, db){
  if(err) throw err;
  db.Collection("employees").fineOne({}, function(err, result){
  if(err) throw err;
  Console.log(result.name);
  db.close();
  });
});
```

open the command terminal and run the following command:

Node Select.js

```
root@aspire-pc: /home/user/Desktop/MongoDatabase
root@aspire-pc:/home/user/Desktop/MongoDatabase# node remove.js
1 record(s) deleted
root@aspire-pc:/home/user/Desktop/MongoDatabase#
```

Select Multiple Records

The find() method is used to select all the records from collection in MongoDB.

Example

Select all records from "employees" collection.

Create a js file named "Selectall.js", having the following code:

```
VarMongo client = require ('mongo DB'). Mongoclient;
Var url = "mongo db://localhost:27017/Mongo Database";
Mongoclient.connect (url, function (err, db) {
  if (err) throw err;
  db.collection ("employees").find({}).to array(function
                                        (err, result) {
    if(err) throw err;
    console.log (result);
    db.close();
  });
});
```

Open the command terminal and run the following command:

```
Node selectall.js
```

Node.js MongoDB Filter Query

The find() method is also used to filter the result on a specific parameter. You can filter the result by using a query object.

Example

Filter the records to retrieve the specific employee whose address is "Delhi".

Create a .js file named "query1.js" having the following code:

```
Var http = require('http');
VarMongoClient = require('mongodb').MongoClient;
Varurl = "mongodb://localhost:27017/MongoDatabase";
MongoClient.connect(url,function(err,db){
if(err) throw err;
Varquery = {address:"Delhi"};
db.collection("employees").find(query).toArray(function
                                (err,result){
if(err) throwerr;
console.log(result);
db.close();
});
});
```

Open the Command terminal and run the following Command

```
Node query1.js
```

```
root@aspire-pc: /home/user/Desktop/MongoDatabase
root@aspire-pc:/home/user/Desktop/MongoDatabase# node query1.js
[ { _id: 591040c52a89e8301bde229a,
    name: 'Ajeet Kumar',
    age: '28',
    address: 'Delhi' } ]
root@aspire-pc:/home/user/Desktop/MongoDatabase#
```

Node.js Mongo DB Sorting

In MongoDB, the sort() method is used for sorting the results in ascending or descending order. The sort() method uses a parameter to define the object sorting order.

Value used for sorting in ascending order:

```
[name:1]
```

Value used for Sorting in descending order:

```
[name:-1]
```

Sort in Ascending Order

Example

Sort the records in ascending order by the name
Create a js file named "Sortasc.js" having the following code:

```
Var http = require('http');
Var Mongo client = require('mongodb').mongo client
Var url = "mongodb://localhost:27017/MongoDatabase";
MongoClient.connect(url, function(err, db) {
if(err) throw err;
Var mysort = {name:1};
db.collection("employees").find().sort(mysort).toArray
                    (function(err, result) {
if(err) throw err;
Console.log(result);
db.close();
});
});
```

Open the Command terminal and run the following comma

```
Node sortasc.js
```

```
root@aspire-pc: /home/user/Desktop/MongoDatabase
root@aspire-pc:/home/user/Desktop/MongoDatabase# node sortasc.js
[ { _id: 591040c52a89e8301bde229a,
    name: 'Ajeet Kumar',
    age: '28',
    address: 'Delhi' },
  { _id: 59104e062f60cd3366ceb7dd,
    name: 'Juck Ross',
    age: '45',
    address: 'London' },
  { _id: 59104e062f60cd3366ceb7da,
    name: 'Mahesh Sharma',
    age: '25',
    address: 'Ghaziabad' },
  { _id: 59104e062f60cd3366ceb7db,
    name: 'Tom Moody',
    age: '31',
    address: 'CA' },
  { _id: 59104e062f60cd3366ceb7dc,
    name: 'Zahira Wasim',
    age: '19',
    address: 'Islamabad' } ]
root@aspire-pc:/home/user/Desktop/MongoDatabase#
```

Node.js Mongo DB Remove

In Mongo DB, you can delete records or documents by using the remove() method. The first parameter of the remove() method is a query object which specifies the document to delete.

Example

Remove the record of employee whose address is Ghaziabad.

Create a js file named "remove.js", having the following code:

```
Var http = require('http');
Var MongoClient=require('mongodb').Mongo client;
Var url = "mongodb://localhost:27017/MongoDatabase";
MongoClient.connect(url, function (err, db){
if (err) throw err;
Var myquery = {address: 'Ghaziabad'};
db.Collection("employees").remove(myquery, function(err, obj){
if (err) throw err;
Console.log(obj.result.n +"record(s) deleted");
db.close();
});
});
```

Open the command terminal and run the following command:

```
Node remove.js
```