

COMPLETE NOTES ON

Node.JS

COPYRIGHTED BY:-

INSTAGRAM:- @Codes.learning

YOUTUBE:- Point of Programming

TELEGRAM:- codeslearningTG

WRITTEN BY:-

Mayuri Kishor Khavche

(Software Engineer)

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 StringDecoder	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. MySQL Drop Table • Node.js MongoDB	75
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 used to run real-time server applications.

The definition given by its official documentation is as follows:

? Node.js is a platform built on Google'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. 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 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 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:

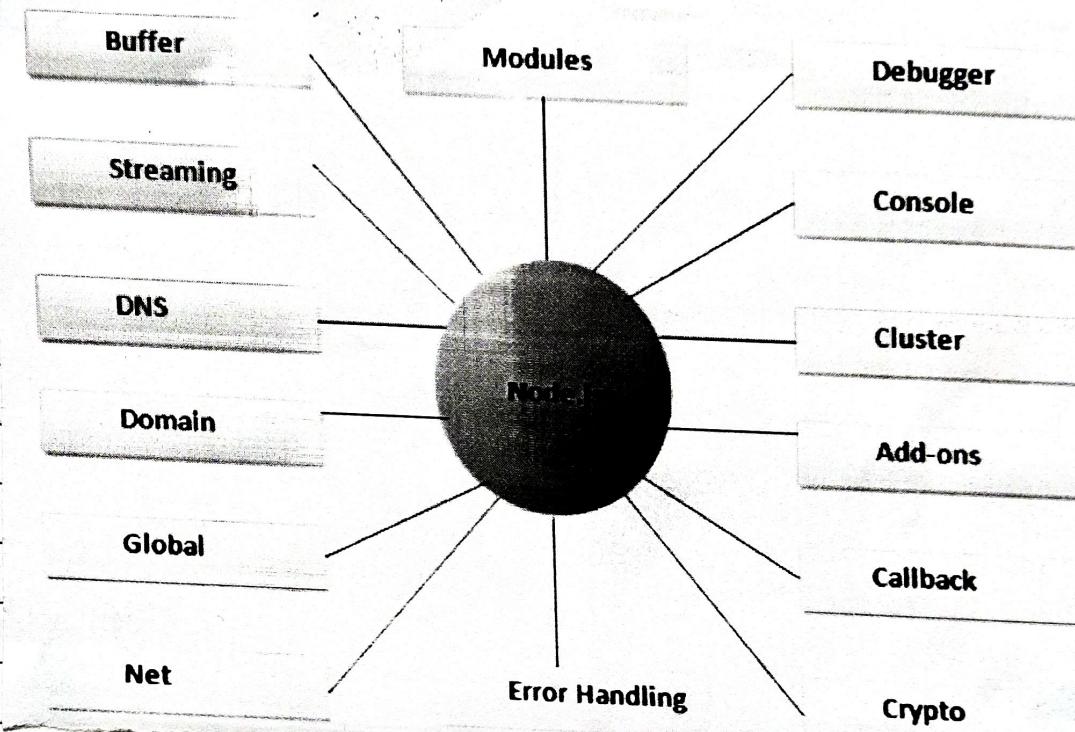
Node.js has an open source community which has produced many excellent modules to add additional capabilities to Node.js applications.

7. License:

Node.js is released under the MIT license.

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 Node.js 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.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:

```
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:

```

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 step 2 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 on 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

```
Node.js command prompt
Your environment has been set up for using Node.js 4.4.2 (x64) and npn.
C:\Users\javatpoint1\Desktop>node console_example1.js
Hello JavaPoint
C:\Users\javatpoint1\Desktop>
```

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

File: Console_example2.js

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

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 command prompt
C:\Users\javatpoint1\Desktop>node console_example3.js
Error: Hell! This is a wrong method.
C:\Users\javatpoint1\Desktop>
```

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 command prompt
C:\Users\javatpoint1\Desktop>node console_example4.js
Don't mess with me John! Don't mess with me!
C:\Users\javatpoint1\Desktop>
```

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 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 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.

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 Java Script.

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\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 terminates the node repl.

3. Ctrl+D:

It terminates 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 disabled 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 exerted. 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

```
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. OS.endianess()

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 function 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' 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():

(1)- 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 timer 1.js

```
setInterval(function(){
```

```
    console.log('SetInterval: Hey! 1 millisecond completed!');
```

```
}, 1000);
```

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

```
node timer 1.js
```

A screenshot of a Windows Command Prompt window titled "Node.js command prompt". The command `node timer1.js` is run, which contains a setInterval loop. The output shows the interval function being triggered every 1 millisecond, printing "Hey! 1 millisecond completed!". The session ends with `Ctrl+C`.

```
C:\Users\jawaatpoint1\Desktop>node timer1.js
Hey! 1 millisecond completed!
Ctrl+C
C:\Users\jawaatpoint1\Desktop>
```

Node.js Errors

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

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

Node.js Errors Example

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

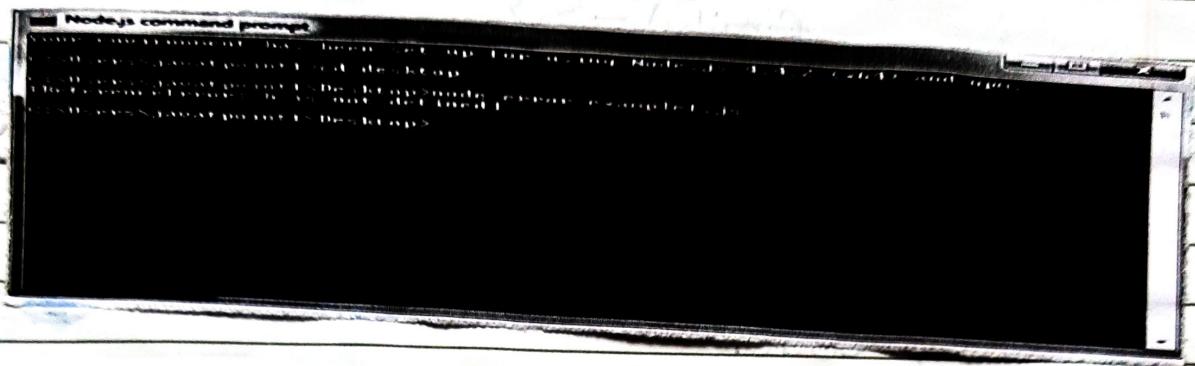
File error_example1.js

```
// 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.resolveName(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

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

A screenshot of a Windows Command Prompt window titled "Node.js command prompt". The window shows the following text:
Your environment has been set up for using Node.js 4.4.2 (x64) and npm.
C:\Users\javatpoint\Desktop>
C:\Users\javatpoint\Desktop>node dns_example1.js
addresses: 144.76.11.18
family: 4
C:\Users\javatpoint\Desktop>

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 ${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
    console.log('Connected to server!');
    client.write('World!\r\n');
}).on('data', (data) => {
    console.log(data.toString());
    client.end();
}).on('end', () => {
    console.log('disconnected from server');
});
```

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

`node netclient.js`

```
C:\Windows\system32\cmd.exe
C:\Users\javatpoint\Desktop>node net_client.js
connected to server!
goodbye

disconnected from server
C:\Users\javatpoint\Desktop>
```

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

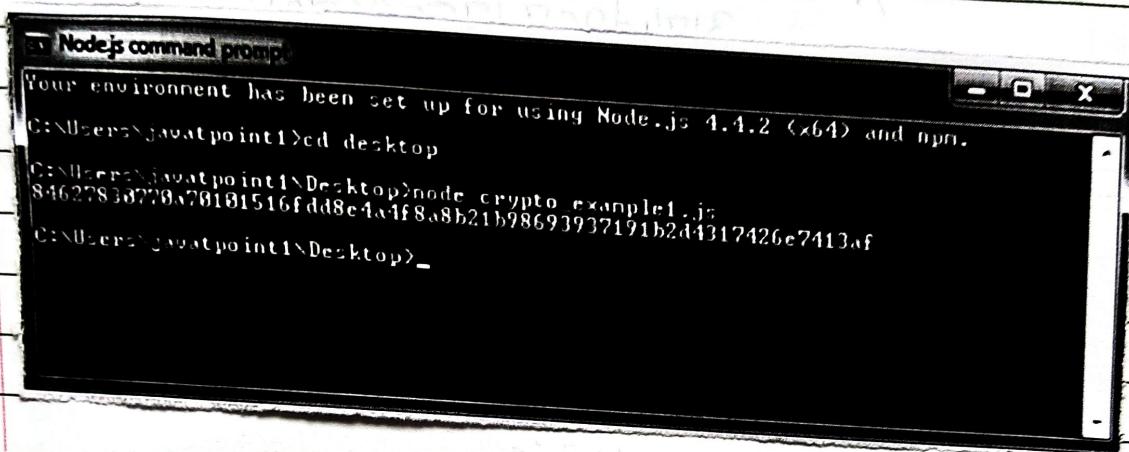
File: `crypto_example1.js`

```
Const Crypto = require('crypto');
Const Secret = 'abcdefg';
```

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

console.log(hash);
```

Open Node.js Command prompt and run the following code
node crypto_example1.js



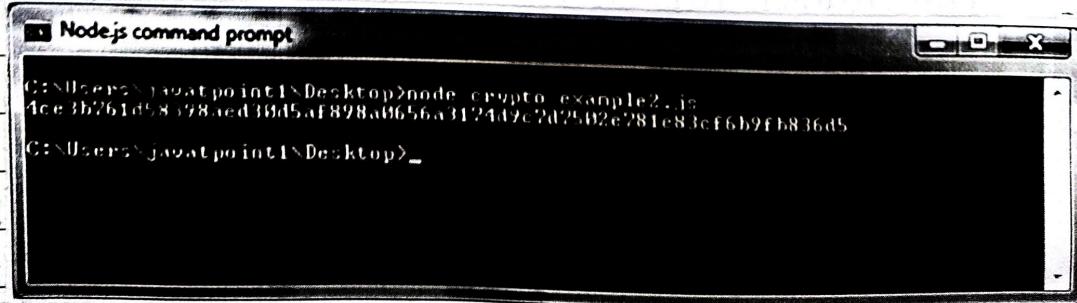
```
Node.js command prompt
Your environment has been set up for using Node.js 4.4.2 (x64) and npm.
C:\Users\javatpoint1>cd desktop
C:\Users\javatpoint1\Desktop>node crypto_example1.js
84627838778a78101516fdd8e4e1f8a8b21b98693937191b2d4317426e7410af
C:\Users\javatpoint1\Desktop>
```

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 code
node crypto_example2.js



```
Node.js command prompt
C:\Users\javatpoint1\Desktop>node crypto_example2.js
4ce3b761d58398a6d30d5af898a0656a312449c7d7502e281e83cf6b9ff836d5
C:\Users\javatpoint1\Desktop>
```

Decryption example using Decipher

File: crypto_example3.js

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

```
Const decipher = crypto.createDecipher('aes192',  
    'a password');
```

```
Var encrypted = '4ce3b761d58398aed30d5af898a065  
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
```

The screenshot shows a Windows command prompt window titled "Node.js command prompt". The command "node crypto_example3.js" is entered, followed by the output "Hello JavaTpoint". The command prompt is located at "C:\Users\javatpoint\Desktop".

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 private key. You have to use `require('tls')` to access this module.

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

Node.js TLS Client example

File: `tls_client.js`

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

```
function Connected(stream) {
```

```
if (stream) {
```

```
// Socket connected
```

```
stream.write("GET /HTTP/1.0\r\nHost: encrypted.  
google.com:443\r\n\r\n\r\n");
```

```
} 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
});

```

The screenshot shows a terminal window titled "Node.js command prompt". The command entered is "node tls-client.js". The output is as follows:

```

C:\Users\javatpoint1\Desktop>node tls-client.js
HTTP/1.0 302 Found
Cache-Control: private
Content-Type: text/html; charset=UTF-8
Location: http://www.google.co.in/?gfe_rd=cr&ei=c0xB09i0D6_T8gf82Y0I
Content-Length: 260
Date: Sun, 22 May 2016 06:06:43 GMT
Alternate-Protocol: 443:quic
Alt-Svc: quic ":443"; ma=2592000; v="34,33,32,31,30,29,28,27,26,25"
<HTML><HEAD><meta http-equiv="content-type" content="text/html; charset=utf-8">
<TITLE>302 Moved</TITLE></HEAD><BODY>
<H1>302 Moved</H1>
The document has moved
<A HREF="http://www.google.co.in/?gfe_rd=cr&ei=c0xB09i0D6_T8gf82Y0I">here</A>
</BODY></HTML>

```

C:\Users\javatpoint1\Desktop>

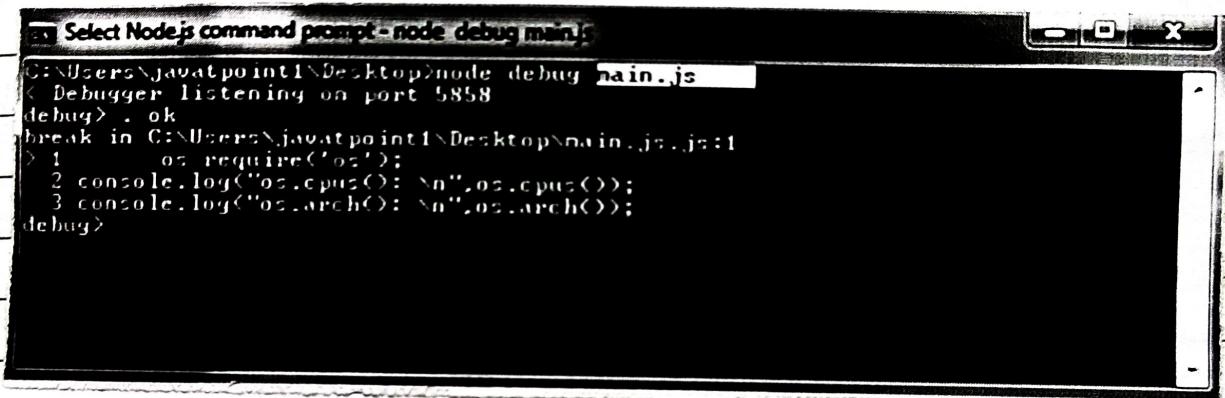
Node.js Debugger

Node.js provides a simple TCP based protocol and built-in debugging client. For debugging your JavaScript file, you can use the 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



Select Node.js command prompt - node debug main.js

```
C:\Users\javatpoint\Desktop>node debug main.js
Debugger listening on port 5858
debug> .ok
break in C:\Users\javatpoint\Desktop\main.js:1
> 1      os=require('os');
  2 console.log("os.cpus(): "+os.cpus());
  3 console.log("os.arch(): "+os.arch());
debug>
```

Node.js Process

Node.js provides the facility to get process information such as process id, architecture, platform, version, release uptime, CPU 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 command 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
versons	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
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

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 childprocess.exec() method

The `childprocess.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 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 childprocess.exec() example
File: childprocess_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[, 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 workerProcess = child_process.spawn('node', [support]);
```

```
  workerProcess.stdout.on('data', function(data) {
```

```
    console.log('stdout:' + data);
  });
}
```

```
workerProcess.stderr.on('data', function(data) {
```

```
  console.log('stderr:' + data);
});
```

Worker Process. On('close', function

 console.log('child process exited with code' + code);

});

}

Node.js childprocess.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 child process instance.

`child_process.fork(modulePath[, args[, 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 WorkerProcess = child_process.fork("support.js", [i])
```

```
    WorkerProcess.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 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 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 syntax 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: abcdefghijklmno
```

```
ABCDEFGHIJKLMNO
```

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 writerStream = fs.createWriteStream ('output.txt');
// Write the data to stream with encoding to be utf-8
writerStream.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 whereas the Synchronous method blocks.

Lets take an example:

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

File: Input.txt

Javaatpoint 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.
wt	open file for reading and writing. the file is created (if it does not exist) or truncated (if it exists).
wxt	like 'wt' 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.
- at 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 successfully!");
});
```

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.