```
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
Worker Process. 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 Workerprocess = 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('/ssit/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 ('7486973206973206120746C3a97374',
                         '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_example1.js

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