Setup

Sunday, November 27, 2016 12:35 PM

Windows Installtion

Download the exe, installtion sets up the

Linux installation

$ cd /tmp
$ wget http://nodejs.org/dist/v6.3.1/node-v6.3.1-linux-x64.tar.gz
$ tar xvfz node-v6.3.1-linux-x64.tar.gz
$ mkdir -p /usr/local/nodejs
$ mv node-v6.3.1-linux-x64/* /usr/local/nodejs

Add /usr/local/nodejs/bin to the PATH

export PATH=$PATH:/usr/local/nodejs/bin

Node Package Manager (NPM) Set up

provides two main functionalities

1. Online repositories for node.js packages/modules which are searchable on search.nodejs.org

1. Command line utility to install Node.js packages, do version management and dependency management of Node.js packages

```
D:\CODE_BASE\samples\node-js\example1>npm -version
3.10.8
```

famous Node.js web framework module called express

npm install express
var express = require('express');

Global vs Local Installation

By default, NPM installs any dependency in the local mode. Here local mode refers to the package installation in node_modules directory lying in the folder where Node application is present. Locally deployed packages are accessible via require() method. For example, when we installed express module, it created node_modules directory in the current directory where it installed the express module.

this command lists the locally installed packages
npm ls

```
D:\CODE_BASE\samples\node-js\example1>npm ls
D:\CODE_BASE\samples\node-js\example1
`-- (empty)
```

Global Installation

$ npm install express -g

$ npm ls -g

```
D:\CODE_BASE\samples\node-js\example1>npm ls -g
C:\Users\development\AppData\Roaming\npm
`-- (empty)
```

Actions on a Package

$ npm update express

$ npm search express

$ npm uninstall express

Creating a Package

$ npm init
$ npm adduser
$ npm publish

Basic Concept

Sunday, November 27, 2016 12:43 PM

Basic Concepts

1. Node.js is a single-threaded application, but it can support concurrency via the concept of event and callbacks.
2. Every API of Node.js is asynchronous and being single-threaded, they use async function calls to maintain concurrency.
3. Node uses observer pattern. Node thread keeps an event loop and whenever a task gets completed, it fires the corresponding event which signals the event-listener function to execute.

What is Callback?

1. Callback is an asynchronous equivalent for a function.
2. A callback function is called at the completion of a given task.
3. Node makes heavy use of callbacks.
4. All the APIs of Node are written in such a way that they support callbacks.

For example, a function to read a file may start reading file and return the control to the execution environment immediately so that the next instruction can be executed. Once file I/O is complete, it will call the callback function while passing the callback function, the content of the file as a parameter. So there is no blocking or wait for File I/O. This makes Node.js highly scalable, as it can process a high number of requests without waiting for any function to return results.

Import required modules — We use the require directive to load Node.js modules.

Create server — A server which will listen to client's requests similar to Apache HTTP Server.

Read request and return response — The server created in an earlier step will read the HTTP request made by the client which can be a browser or a console and return the response

Step 1 - Import Required Module

We use the require directive to load the http module and store the returned HTTP instance into an http variable as follows
var http = require("http");

Step 2 - Create Server

We use the created http instance and call http.createServer() method to create a server instance and then we bind it at port 8081 using the listen method associated with the server instance. Pass it a function with parameters request and response. Write the sample implementation to always return "Hello World".

```
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(1011);

// Console will print the message
console.log('Server running at http://127.0.0.1:1011/');
```

#run this program using
node main.js

Buffer

Sunday, November 27, 2016 7:49 PM

Buffer class is a global class that can be accessed in an application without importing the buffer module

Pure JavaScript is Unicode friendly, but it is not so for binary data. While dealing with TCP streams or the file system, it's necessary to handle octet streams.
Node provides Buffer class which provides instances to store raw data similar to an array of integers but corresponds to a raw memory allocation outside the V8 heap.

Creating Buffer

Syntax	Description
var buf = new Buffer(10);	syntax to create an uninitiated Buffer of 10 octets
var buf = new Buffer([10, 20, 30, 40, 50]);	syntax to create a Buffer from a given array
var buf = new Buffer("Simply Easy Learning", "utf-8");	syntax to create a Buffer from a given string and optionally encoding type • uft-8 is the default • ascii • utf16le • ucs2 • base64 • hex

Writing to a Buffer

buf.write(string[, offset][, length][, encoding])	Parameters *string — This is the string data to be written to buffer. offset — This is the index of the buffer to start writing at. Default value is 0.(optional) length — This is the number of bytes to write. Defaults to buffer.length.(optional) encoding — Encoding to use. 'utf8' is the default encoding.(optional) Return Value This method returns the number of octets written. If there is not enough space in the buffer to fit the entire string, it will write a part of the string

Example

```
buf = new Buffer(256);
len = buf.write("Simply Easy Learning");

console.log("Octets written : "+  len);
```

Reading a buffer

buf.toString([encoding][, start][, end])	Parameters encoding — Encoding to use. 'utf8' is the default encoding. start — Beginning index to start reading, defaults to 0. end — End index to end reading, defaults is complete buffer. Return Value This method decodes and returns a string from buffer data encoded using the specified character set encoding.

Example

```
buf = new Buffer(26);
for (var i = 0 ; i < 26 ; i++) {
  buf[i] = i + 97;
}

console.log( buf.toString('ascii'));       // outputs: abcdefghijklmnopqrstuvwxyz
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 defaults to 'utf8', outputs abcde

Syntax	Parameters and Return Value	Description

buf.toJSON()		Convert to a JSON string
		var buf = new Buffer('Simply Easy Learning');
		var json = buf.toJSON(buf);
		
		console.log(json);

Buffer.concat(list[, totalLength])	Parameters
	Here is the description of the parameters used −
	
	list − Array List of Buffer objects to be concatenated.
	
	totalLength − This is the total length of the buffers when concatenated.
	
	Return Value
	This method returns a Buffer instance.
		syntax of the method to concatenate Node buffers to a single Node Buffer
		var buffer1 = new Buffer('TutorialsPoint ');
		var buffer2 = new Buffer('Simply Easy Learning');
		var buffer3 = Buffer.concat([buffer1,buffer2]);
		console.log("buffer3 content: " + buffer3.toString());
		
		//buffer3 content: TutorialsPoint Simply Easy Learning

buf.compare(otherBuffer);	Parameters
	Here is the description of the parameters used −
	
	otherBuffer − This is the other buffer which will be compared with buf
	
	Return Value
	Returns a number indicating whether it comes before or after or is the same as the otherBuffer in sort order.
		syntax of the method to compare two Node buffers
		Example
		var buffer1 = new Buffer('ABC');
		var buffer2 = new Buffer('ABCD');
		var result = buffer1.compare(buffer2);
		
		if(result < 0) {
		  console.log(buffer1 +" comes before " + buffer2);
		}else if(result == 0){
		  console.log(buffer1 +" is same as " + buffer2);
		}else {
		  console.log(buffer1 +" comes after " + buffer2);
		}
		
		//ABC comes before ABCD

buf.copy(targetBuffer[, targetStart][, sourceStart][, sourceEnd])	Parameters
	targetBuffer − Buffer object where buffer will be copied.
	
	targetStart − Number, Optional, Default: 0
	
	sourceStart − Number, Optional, Default: 0
	
	sourceEnd − Number, Optional, Default: buffer.length
	
	Return Value
	No return value. Copies data from a region of this buffer to a region in the target buffer even if the target memory region overlaps with the source. If undefined, the targetStart and sourceStart parameters default to 0, while sourceEnd defaults to buffer.length.
		syntax of the method to copy a node buffer
		
		Example
		var buffer1 = new Buffer('ABC');
		
		//copy a buffer
		var buffer2 = new Buffer(3);
		buffer1.copy(buffer2);
		console.log("buffer2 content: " + buffer2.toString());
		
		//buffer2 content: ABC

buf.slice([start][, end])	Parameters start − Number, Optional, Default: 0 end − Number, Optional, Default: buffer.length	syntax of the method to get a sub-buffer of a node buffer var buffer1 = new

	Return Value
	Returns a new buffer which references the same memory as the old one, but offset and cropped by the start (defaults to 0) and end (defaults to buffer.length) indexes. Negative indexes start from the end of the buffer
		Buffer('StringTest');
		
		//slicing a buffer
		var buffer2 = buffer1.slice(0,6);
		console.log("buffer2 content: " + buffer2.toString());
		
		buffer2 content: String

buf.length;	Return Value Returns the size of a buffer in bytes.	

Class Methods

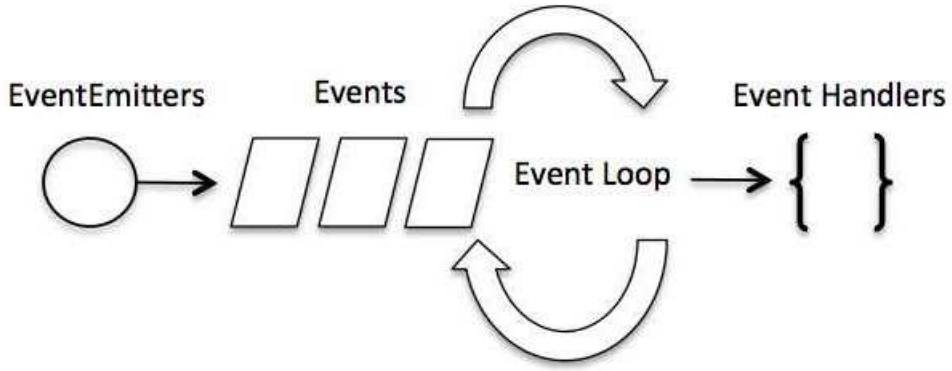
Method & Description
Buffer.isEncoding(encoding) Returns true if the encoding is a valid encoding argument, false otherwise.
Buffer.isBuffer(obj) Tests if obj is a Buffer.
Buffer.byteLength(string[, encoding]) Gives the actual byte length of a string. encoding defaults to 'utf8'. It is not the same as String.prototype.length, since String.prototype.length returns the number of characters in a string.
Buffer.concat(list[, totalLength]) Returns a buffer which is the result of concatenating all the buffers in the list together.
Buffer.compare(buf1, buf2) The same as buf1.compare(buf2). Useful for sorting an array of buffers.

Event Driven Programing

Sunday, November 27, 2016 1:17 PM

Event-Driven Programming

Node.js uses events heavily and it is also one of the reasons why Node.js is pretty fast compared to other similar technologies.
As soon as Node starts its server, it simply initiates its variables, declares functions and then simply waits for the event to occur.



In an event-driven application, there is generally a main loop that listens for events, and then triggers a callback function when one of those events is detected.

Events

```
// Import events module
var events = require('events');
```

1. Although events look quite similar to callbacks, the difference lies in the fact that callback functions are called when an asynchronous function returns its result, whereas event handling works on the observer pattern.
2. The functions that listen to events act as Observers.
3. Whenever an event gets fired, its listener function starts executing.
4. Node.js has multiple in-built events available through events module and EventEmitter class which are used to bind events and event-listeners

EventEmitter

Class Methods

Method & Description
listenerCount(emitter, event) Returns the number of listeners for a given event.

Events

Events & Description
newListener • **event** – String: the event name • **listener** – Function: the event handler function This event is emitted any time a listener is added. When this event is triggered, the listener may not yet have been added to the array of listeners for the event.
removeListener • **event** – String The event name

• **listener** — Function The event handler function

This event is emitted any time someone removes a listener. When this event is triggered, the listener may not yet have been removed from the array of listeners for the event.

EventEmitter class lies in the events module. It is accessible via the following code

```
// Import events module
var events = require('events');

// Create an eventEmitter object
var eventEmitter = new events.EventEmitter();
```

```
// 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 function
eventEmitter.on('data_received', function(){
  console.log('data received succesfully.');
});

// Fire the connection event
eventEmitter.emit('connection');

console.log("Program Ended.");
```

When a new listener is added, 'newListener' event is fired and when a listener is removed, 'removeListener' event is fired.
When an EventEmitter instance faces any error, it emits an 'error' event.
EventEmitter provides multiple properties like on and emit. on property is used to bind a function with the event and emit is used to fire an event.

Method & Description
addListener(event, listener) Adds a listener at the end of the listeners array for the specified event. No checks are made to see if the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be
on(event, listener) Adds a listener at the end of the listeners array for the specified event. No checks are made to see if

the listener has already been added. Multiple calls passing the same combination of event and listener will result in the listener being added multiple times. Returns emitter, so calls can be
once(event, listener) Adds a one time listener to the event. This listener is invoked only the next time the event is fired, after which it is removed. Returns emitter, so calls can be chained.
removeListener(event, listener) Removes a listener from the listener array for the specified event. **Caution —** It changes the array indices in the listener array behind the listener. removeListener will remove, at most, one instance of a listener from the listener array. If any single listener has been added multiple times to the listener array for the specified event, then removeListener must be called multiple times to remove
removeAllListeners([event]) Removes all listeners, or those of the specified event. It's not a good idea to remove listeners that were added elsewhere in the code, especially when it's on an emitter that you didn't create (e.g. sockets or file streams). Returns emitter, so calls can be chained.
setMaxListeners(n) By default, EventEmitters will print a warning if more than 10 listeners are added for a particular event. This is a useful default which helps finding memory leaks. Obviously not all Emitters should be limited to 10. This function allows that to be increased. Set to zero for unlimited.
listeners(event) Returns an array of listeners for the specified event.
emit(event, [arg1], [arg2], [...]) Execute each of the listeners in order with the supplied arguments. Returns true if the event had listeners, false otherwise.

Strems

Sunday, November 27, 2016 10:12 PM

Streams are objects that let you read data from a source or write data to a destination in continuous fashion.

In Node.js, there are four types of streams

- **Readable** – Stream which is used for read operation.
- **Writable** – Stream which is used for write operation.
- **Duplex** – Stream which can be used for both read and write operation.
- **Transform** – A type of duplex stream where the output is computed based on input.

Each type of Stream is an **EventEmitter** instance and throws several events at different instance of times. For example, some of the commonly used events are –

- **data** – This event is fired when there is data is available to read.
- **end** – This event is fired when there is no more data to read.
- **error** – This event is fired when there is any error receiving or writing data.
- **finish** – This event is fired when all the data has been flushed to underlying system.

Reading from a Stream

```
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");
```

```
$ node main.js
```

Writing to a Stream

```
var fs = require("fs");
var data = 'Simply Easy Learning';
// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');
// Write the data to stream with encoding to be utf8
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(err){
  console.log(err.stack);
});

console.log("Program Ended");
```

Now run the main.js to see the result −

```
$ node main.js
```

Piping the Streams

Piping is a mechanism where we provide the output of one stream as the input to another stream.

It is normally used to get data from one stream and to pass the output of that stream to another stream.

There is no limit on piping operations.

```
var fs = require("fs");
// Create a readable stream
var readerStream = fs.createReadStream('input.txt');
// Create a writable stream
var writerStream = fs.createWriteStream('output.txt');
// Pipe the read and write operations
// read input.txt and write data to output.txt
readerStream.pipe(writerStream);
console.log("Program Ended");

$ node main.js
```

Chaining the Streams

Chaining is a mechanism to connect the output of one stream to another stream and create a chain of multiple stream operations. It is normally used with piping operations. Now we'll use piping and chaining to first compress a file and then decompress the same.

Create a js file named main.js with the following code −

```
var fs = require("fs");
var zlib = require('zlib');

// Compress the file input.txt to input.txt.gz
fs.createReadStream('input.txt')
  .pipe(zlib.createGzip())
  .pipe(fs.createWriteStream('input.txt.gz'));
console.log("File Compressed.");

$ node main.js
```

You will find that input.txt has been compressed and it created a file input.txt.gz in the current directory.

Decompress

```
var fs = require("fs");
var zlib = require('zlib');
 // Decompress the file input.txt.gz to input.txt
fs.createReadStream('input.txt.gz')
  .pipe(zlib.createGunzip())
```

```
  .pipe(fs.createWriteStream('input.txt'));
console.log("File Decompressed.");
```

```
$ node main.js
```

Files

Sunday, November 27, 2016 11:42 PM

Node implements File I/O using simple wrappers around standard POSIX functions.

The Node File System (fs) module can be imported using the following syntax

var fs = require("fs")

Synchronous vs Asynchronous

Every method in the fs module has synchronous as well as asynchronous forms.

Asynchronous methods take the last parameter as the completion function callback.

And the first parameter of the callback function as error.

It is better to use an asynchronous method instead of a synchronous method, as the former never blocks a program during its execution, whereas the second one does.

Example

```
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");
```

Now run the main.js to see the result –

```
$ node main.js
```

Open a File

Syntax		
fs.open(path, flags[, mode], callback)	Parameters Here is the description of the parameters used – • **path** – This is the string having file name including path. • **flags** – Flags indicate the behavior of the file to be opened. All possible values have been mentioned below. • **mode** – It sets the file mode (permission and sticky bits), but only if the file was created. It defaults to 0666, readable and writeable. • **callback** – This is the callback function which gets two arguments (err, fd).	syntax of the method to open a file in asynchronous mode

Flags

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, asking 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 the 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 the path exists.
a+	Open file for reading and appending. The file is created if it does not exist.
ax+	Like 'a+' but fails if the the path exists.

Example

open a file input.txt for reading and writing.

```
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!");
});
```

Now run the main.js to see the result −

```
$ node main.js
```

Verify the Output.

```
Going to open file!
File opened successfully!
```

Get File Information

Syntax	Parameters	
fs.stat(path, callback)	Here is the description of the parameters used − • **path** − This is the string having file name including path. • **callback** − This is the callback function which gets two arguments (err, stats) where **stats** is an object of fs.Stats type which is printed below in the example.	syntax of the method to get the information about a file

Apart from the important attributes which are printed below in the example, there are several useful methods available in **fs.Stats** class which can be used to check file type. These methods are given in the following table.

Method	Description
stats.isFile()	Returns true if file type of a simple file.
stats.isDirectory()	Returns true if file type of a directory.
stats.isBlockDevice()	Returns true if file type of a block device.
stats.isCharacterDevice()	Returns true if file type of a character device.
stats.isSymbolicLink()	Returns true if file type of a symbolic link.
stats.isFIFO()	Returns true if file type of a FIFO.
stats.isSocket()	Returns true if file type of asocket.

Example

```
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("isDirectory ? " + stats.isDirectory());
});
```

Now run the main.js to see the result −

```
$ node main.js
```

Verify the Output.

```
Going to get file info!
{
  dev: 1792,
  mode: 33188,
  nlink: 1,
  uid: 48,
  gid: 48,
  rdev: 0,
  blksize: 4096,
  ino: 4318127,
  size: 97,
  blocks: 8,
  atime: Sun Mar 22 2015 13:40:00 GMT-0500 (CDT),
  mtime: Sun Mar 22 2015 13:40:57 GMT-0500 (CDT),
  ctime: Sun Mar 22 2015 13:40:57 GMT-0500 (CDT)
}
Got file info successfully!
isFile ? true
isDirectory ? false
```

Writing a File

Syntax

Following is the syntax of one of the methods to write into a file −

```
fs.writeFile(filename, data[, options], callback)
```

This method will over-write the file if the file already exists. If you want to write into an existing file then you should use another method available.

Parameters

Here is the description of the parameters used −

- **path** − This is the string having the file name including path.
- **data** − This is the String or Buffer to be written into the file.

- **options** – The third parameter is an object which will hold {encoding, mode, flag}. By default. encoding is utf8, mode is octal value 0666. and flag is 'w'
- **callback** – This is the callback function which gets a single parameter err that returns an error in case of any writing error.

Example

Let us create a js file named **main.js** having the following code –

```
var fs = require("fs");


console.log("Going to write into existing file");
fs.writeFile('input.txt', 'Simply Easy Learning!',  function(err) {
  if (err) {
    return console.error(err);
  }

  console.log("Data written successfully!");
  console.log("Let's read newly written data");
  fs.readFile('input.txt', function (err, data) {
    if (err) {
      return console.error(err);
    }
    console.log("Asynchronous read: " + data.toString());
  });
});
```

Now run the main.js to see the result –

```
$ node main.js
```

Verify the Output.

```
Going to write into existing file
Data written successfully!
Let's read newly written data
Asynchronous read: Simply Easy Learning!
```

Reading a File

Syntax

Following is the syntax of one of the methods to read from a file –

```
fs.read(fd, buffer, offset, length, position, callback)
```

This method will use file descriptor to read the file. If you want to read the file directly using the file name, then you should use another method available.

Parameters

Here is the description of the parameters used –

- **fd** – This is the file descriptor returned by fs.open().
- **buffer** – This is the buffer that the data will be written to.
- **offset** – This is the offset in the buffer to start writing at.
- **length** – This is an integer specifying the number of bytes to read.
- **position** – This is an integer specifying where to begin reading from in the file. If position is null, data will be read from the current file position.
- **callback** – This is the callback function which gets the three arguments, (err, bytesRead, buffer).

Example

Let us create a js file named **main.js** with the following code −

```
var fs = require("fs");
var buf = new Buffer(1024);

console.log("Going to open an existing file");
fs.open('input.txt', 'r+', function(err, fd) {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
  console.log("Going to read the file");
  fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
    if (err){
      console.log(err);
    }
    console.log(bytes + " bytes read");

    // Print only read bytes to avoid junk.
    if(bytes > 0){
      console.log(buf.slice(0, bytes).toString());
    }
  });
});
```

Now run the main.js to see the result −

```
$ node main.js
```

Verify the Output.

```
Going to open an existing file
File opened successfully!
Going to read the file
97 bytes read
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

Closing a File

Syntax

Following is the syntax to close an opened file −

```
fs.close(fd, callback)
```

Parameters

Here is the description of the parameters used −

- **fd** − This is the file descriptor returned by file fs.open() method.
- **callback** − This is the callback function No arguments other than a possible exception are given to the completion callback.

Example

Let us create a js file named **main.js** having the following code −

```
var fs = require("fs");
var buf = new Buffer(1024);
```

```
console.log("Going to open an existing file");
fs.open('input.txt', 'r+', function(err, fd) {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
  console.log("Going to read the file");

  fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
    if (err){
      console.log(err);
    }

    // Print only read bytes to avoid junk.
    if(bytes > 0){
      console.log(buf.slice(0, bytes).toString());
    }

    // Close the opened file.
    fs.close(fd, function(err){
      if (err){
        console.log(err);
      }
      console.log("File closed successfully.");
    });
  });
});
```

Now run the main.js to see the result −

```
$ node main.js
```

Verify the Output.

```
Going to open an existing file
File opened successfully!
Going to read the file
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!

File closed successfully.
```

Truncate a File

Syntax

Following is the syntax of the method to truncate an opened file −

```
fs.ftruncate(fd, len, callback)
```

Parameters

Here is the description of the parameters used −

- **fd** – This is the file descriptor returned by fs.open().
- **len** – This is the length of the file after which the file will be truncated.
- **callback** – This is the callback function No arguments other than a possible exception are given to the completion callback.

Example

Let us create a js file named **main.js** having the following code –

```
var fs = require("fs");
var buf = new Buffer(1024);

console.log("Going to open an existing file");
fs.open('input.txt', 'r+', function(err, fd) {
  if (err) {
    return console.error(err);
  }
  console.log("File opened successfully!");
  console.log("Going to truncate the file after 10 bytes");

  // Truncate the opened file.
  fs.ftruncate(fd, 10, function(err){
    if (err){
      console.log(err);
    }
    console.log("File truncated successfully.");
    console.log("Going to read the same file");

    fs.read(fd, buf, 0, buf.length, 0, function(err, bytes){
      if (err){
        console.log(err);
      }

      // Print only read bytes to avoid junk.
      if(bytes > 0){
        console.log(buf.slice(0, bytes).toString());
      }

      // Close the opened file.
      fs.close(fd, function(err){
        if (err){
          console.log(err);
        }
        console.log("File closed successfully.");
      });
    });
  });
});
```

Now run the main.js to see the result −

```
$ node main.js
```

Verify the Output.

```
Going to open an existing file
File opened successfully!
Going to truncate the file after 10 bytes
File truncated successfully.
Going to read the same file
Tutorials
File closed successfully.
```

Delete a File

Syntax

Following is the syntax of the method to delete a file −

```
fs.unlink(path, callback)
```

Parameters

Here is the description of the parameters used −

- **path** − This is the file name including path.
- **callback** − This is the callback function No arguments other than a possible exception are given to the completion callback.

Example

Let us create a js file named **main.js** having the following code −

```
var fs = require("fs");

console.log("Going to delete an existing file");
fs.unlink('input.txt', function(err) {
  if (err) {
    return console.error(err);
  }
  console.log("File deleted successfully!");
});
```

Now run the main.js to see the result −

```
$ node main.js
```

Verify the Output.

```
Going to delete an existing file
File deleted successfully!
```

Create a Directory

Syntax

Following is the syntax of the method to create a directory −

```
fs.mkdir(path[, mode], callback)
```

Parameters

Here is the description of the parameters used −

- **path** − This is the directory name including path.
- **mode** − This is the directory permission to be set. Defaults to 0777.
- **callback** − This is the callback function No arguments other than a possible exception are given to the completion callback.

Example

Let us create a js file named **main.js** having the following code −

```
var fs = require("fs");

console.log("Going to create directory /tmp/test");
fs.mkdir('/tmp/test',function(err){
  if (err) {
    return console.error(err);
  }
  console.log("Directory created successfully!");
});
```

Now run the main.js to see the result −

```
$ node main.js
```

Verify the Output.

```
Going to create directory /tmp/test
Directory created successfully!
```

Read a Directory

Syntax

Following is the syntax of the method to read a directory −

```
fs.readdir(path, callback)
```

Parameters

Here is the description of the parameters used −

- **path** − This is the directory name including path.
- **callback** − This is the callback function which gets two arguments (err, files) where files is an array of the names of the files in the directory excluding '.' and '..'.

Example

Let us create a js file named **main.js** having the following code −

```
var fs = require("fs");

console.log("Going to read directory /tmp");
fs.readdir("/tmp/",function(err, files){
  if (err) {
    return console.error(err);
  }
  files.forEach( function (file){
    console.log( file );
  });
});
```

Now run the main.js to see the result −

```
$ node main.js
```

Verify the Output.

```
Going to read directory /tmp
ccmzx99o.out
ccyCSbkF.out
employee.ser
```

hsperfdata_apache

test

test.txt

Remove a Directory

Syntax

Following is the syntax of the method to remove a directory −

fs.rmdir(path, callback)

Parameters

Here is the description of the parameters used −

- **path** − This is the directory name including path.
- **callback** − This is the callback function No argume nts other than a possible exception are given to the completion callback.

Example

Let us create a js file named **main.js** having the following code −

```
var fs = require("fs");

console.log("Going to delete directory /tmp/test");
fs.rmdir("/tmp/test",function(err){
  if (err) {
    return console.error(err);
  }
  console.log("Going to read directory /tmp");

  fs.readdir("/tmp/",function(err, files){
    if (err) {
      return console.error(err);
    }
    files.forEach( function (file){
      console.log( file );
    });
  });
});
```

Now run the main.js to see the result −

```
$ node main.js
```

Verify the Output.

```
Going to read directory /tmp
ccmzx99o.out
ccyCSbkF.out
employee.ser
hsperfdata_apache
test.txt
```

Methods Reference

Following is a reference of File System module available in Node.js. For more detail you can refer to the official documentation.