NODE.JS - FILE SYSTEM

Node implements File I/O using simple wrappers around standard POSIX functions. Node File System *fs* module can be imported using following syntax:

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

Synchronous vs Asynchronous

Every method in fs module have synchronous as well as asynchronous form. Asynchronous methods takes a last parameter as completion function callback and first parameter of the callback function is error. It is preferred to use asynchronous method instead of synchronous method as former never block the program execution where as the second one does.

Example

Create a text file named **input.txt** having following content

```
Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

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

```
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
```

Verify the Output

```
Synchronous read: Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!

Program Ended
Asynchronous read: Tutorials Point is giving self learning content
to teach the world in simple and easy way!!!!!
```

Following section will give good examples on major File I/O methods.

Open a File

Syntax

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

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

Parameters

Here is the description of the parameters used:

- **path** - This is string having file name including path.
- **flags** - Flag tells the behavior of the file to be opened. All possible values have been mentioned below.
- **mode** - This sets the file mode *permissionandstickybits*, 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*.

Flags

Flags for read/write operations are:

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 *ifitdoesnotexist* or truncated *ifitexists*.
wx	Like 'w' but fails if path exists.
w+	Open file for reading and writing. The file is created *ifitdoesnotexist* or truncated *ifitexists*.
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+	Like 'a+' but fails if path exists.

Example

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

Following is the syntax of the method to get the information about a file:

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

Parameters

Here is the description of the parameters used:

- **path** - This is string having file name including path.
- **callback** - This is the callback function which gets two arguments *err, stats* where **stats** is an object of fs.Stats type which is printed below in the example.

Apart from the important attributes which are printed below in the example, there are number of 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

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

```
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 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 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 string having 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 and used to to return 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 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 file using file name directly then you should use another method available.

Parameters

Here is the description of the parameters used:

- **fd** - This is the file descriptor returned by file fs.open method.
- **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** 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);
```

```
      }
      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 File

Syntax

Following is the syntax of one of the methods 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 which gets 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 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 file fs.open method.
- **len** - This is the length of the file after which file will be truncated.
- **callback** - This is the callback function which gets 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 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 which gets 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 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 which gets 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 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 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 which gets 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 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 a further detail you can refer to official documentation.

SN	Method & Description
1	**fs.rename*oldPath, newPath, callback*** Asynchronous rename. No arguments other than a possible exception are given to the completion callback.
2	**fs.ftruncate*fd, len, callback*** Asynchronous ftruncate. No arguments other than a possible exception are given to the completion callback.
3	**fs.ftruncateSync*fd, len*** Synchronous ftruncate
4	**fs.truncate*path, len, callback*** Asynchronous truncate. No arguments other than a possible exception are given to the completion callback.
5	**fs.truncateSync*path, len*** Synchronous truncate
6	**fs.chown*path, uid, gid, callback*** Asynchronous chown. No arguments other than a possible exception are given to the completion callback.
7	**fs.chownSync*path, uid, gid*** Synchronous chown
8	**fs.fchown*fd, uid, gid, callback*** Asynchronous fchown. No arguments other than a possible exception are given to the completion callback.
9	**fs.fchownSync*fd, uid, gid*** Synchronous fchown

10	**fs.lchown***path, uid, gid, callback* Asynchronous lchown. No arguments other than a possible exception are given to the completion callback.
11	**fs.lchownSync***path, uid, gid* Synchronous lchown
12	**fs.chmod***path, mode, callback* Asynchronous chmod. No arguments other than a possible exception are given to the completion callback.
13	**fs.chmodSync***path, mode* Synchronous chmod.
14	**fs.fchmod***fd, mode, callback* Asynchronous fchmod. No arguments other than a possible exception are given to the completion callback.
15	**fs.fchmodSync***fd, mode* Synchronous fchmod.
16	**fs.lchmod***path, mode, callback* Asynchronous lchmod. No arguments other than a possible exception are given to the completion callback.Only available on Mac OS X.
17	**fs.lchmodSync***path, mode* Synchronous lchmod.
18	**fs.stat***path, callback* Asynchronous stat. The callback gets two arguments *err, stats* where stats is a fs.Stats object.
19	**fs.lstat***path, callback* Asynchronous lstat. The callback gets two arguments *err, stats* where stats is a fs.Stats object. lstat is identical to stat, except that if path is a symbolic link, then the link itself is stat-ed, not the file that it refers to.
20	**fs.fstat***fd, callback* Asynchronous fstat. The callback gets two arguments *err, stats* where stats is a fs.Stats object. fstat is identical to stat, except that the file to be stat-ed is specified by the file descriptor fd.
21	**fs.statSync***path* Synchronous stat. Returns an instance of fs.Stats.
22	**fs.lstatSync***path* Synchronous lstat. Returns an instance of fs.Stats.
23	**fs.fstatSync***fd* Synchronous fstat. Returns an instance of fs.Stats.
24	**fs.link***srcpath, dstpath, callback* Asynchronous link. No arguments other than a possible exception are given to the completion callback.
25	**fs.linkSync***srcpath, dstpath* Synchronous link.
26	**fs.symlink***srcpath, dstpath*[, *type*], *callback* Asynchronous symlink. No arguments other than a possible exception are given to the completion callback. The type argument can be set to 'dir', 'file', or 'junction' *defaultis'file'* and is only available on Windows *ignoredonotherplatforms*. Note that Windows junction points require the destination path to be absolute. When using 'junction', the destination argument will automatically be normalized to absolute path.
27	**fs.symlinkSync***srcpath, dstpath*[, *type*]

	Synchronous symlink.
28	**fs.readlink*path, callback*** Asynchronous readlink. The callback gets two arguments *err, linkString*.
29	**fs.realpath*path*[, *cache*], *callback*** Asynchronous realpath. The callback gets two arguments *err, resolvedPath*. May use process.cwd to resolve relative paths. cache is an object literal of mapped paths that can be used to force a specific path resolution or avoid additional fs.stat calls for known real paths.
30	**fs.realpathSync*path*[, *cache*]** Synchronous realpath. Returns the resolved path.
31	**fs.unlink*path, callback*** Asynchronous unlink. No arguments other than a possible exception are given to the completion callback.
32	**fs.unlinkSync*path*** Synchronous unlink.
33	**fs.rmdir*path, callback*** Asynchronous rmdir. No arguments other than a possible exception are given to the completion callback.
34	**fs.rmdirSync*path*** Synchronous rmdir.
35	**fs.mkdir*path*[, *mode*], *callback*** SAsynchronous mkdir2. No arguments other than a possible exception are given to the completion callback. mode defaults to 0777.
36	**fs.mkdirSync*path*[, *mode*]** Synchronous mkdir.
37	**fs.readdir*path, callback*** Asynchronous readdir3. Reads the contents of a directory. The callback gets two arguments *err, files* where files is an array of the names of the files in the directory excluding '.' and '..'.
38	**fs.readdirSync*path*** Synchronous readdir. Returns an array of filenames excluding '.' and '..'.
39	**fs.close*fd, callback*** Asynchronous close. No arguments other than a possible exception are given to the completion callback.
40	**fs.closeSync*fd*** Synchronous close.
41	**fs.open*path, flags*[, *mode*], *callback*** Asynchronous file open.
42	**fs.openSync*path, flags*[, *mode*]** Synchronous version of fs.open.
43	**fs.utimes*path, atime, mtime, callback***
44	**fs.utimesSync*path, atime, mtime*** Change file timestamps of the file referenced by the supplied path.
45	**fs.futimes*fd, atime, mtime, callback***
46	**fs.futimesSync*fd, atime, mtime***

	Change the file timestamps of a file referenced by the supplied file descriptor.
47	**fs.fsync*fd, callback*** Asynchronous fsync. No arguments other than a possible exception are given to the completion callback.
48	**fs.fsyncSync*fd*** Synchronous fsync.
49	**fs.write*fd, buffer, offset, length*[*, position*]*, callback*** Write buffer to the file specified by fd.
50	**fs.write*fd, data*[*, position*[*, encoding*]]*, callback*** Write data to the file specified by fd. If data is not a Buffer instance then the value will be coerced to a string.
51	**fs.writeSync*fd, buffer, offset, length*[*, position*]** Synchronous versions of fs.write. Returns the number of bytes written.
52	**fs.writeSync*fd, data*[*, position*[*, encoding*]]** Synchronous versions of fs.write. Returns the number of bytes written.
53	**fs.read*fd, buffer, offset, length, position, callback*** Read data from the file specified by fd.
54	**fs.readSync*fd, buffer, offset, length, position*** Synchronous version of fs.read. Returns the number of bytesRead.
55	**fs.readFile*filename*[*, options*]*, callback*** Asynchronously reads the entire contents of a file.
56	**fs.readFileSync*filename*[*, options*]** Synchronous version of fs.readFile. Returns the contents of the filename.
57	**fs.writeFile*filename, data*[*, options*]*, callback*** Asynchronously writes data to a file, replacing the file if it already exists. data can be a string or a buffer.
58	**fs.writeFileSync*filename, data*[*, options*]** The synchronous version of fs.writeFile.
59	**fs.appendFile*filename, data*[*, options*]*, callback*** Asynchronously append data to a file, creating the file if it not yet exists. data can be a string or a buffer.
60	**fs.appendFileSync*filename, data*[*, options*]** The synchronous version of fs.appendFile.
61	**fs.watchFile*filename*[*, options*]*, listener*** Watch for changes on filename. The callback listener will be called each time the file is accessed.
62	**fs.unwatchFile*filename*[*, listener*]** Stop watching for changes on filename. If listener is specified, only that particular listener is removed. Otherwise, all listeners are removed and you have effectively stopped watching filename.
63	**fs.watch*filename*[*, options*][*, listener*]** Watch for changes on filename, where filename is either a file or a directory. The returned object is a fs.FSWatcher.
64	**fs.exists*path, callback*** Test whether or not the given path exists by checking with the file system. Then call the callback argument with either true or false.
65	**fs.existsSync*path***

	Synchronous version of fs.exists.
66	**fs.access***path[, mode], callback* Tests a user's permissions for the file specified by path. mode is an optional integer that specifies the accessibility checks to be performed.
67	**fs.accessSync***path[, mode]* Synchronous version of fs.access. This throws if any accessibility checks fail, and does nothing otherwise.
68	**fs.createReadStream***path[, options]* Returns a new ReadStream object.
69	**fs.createWriteStream***path[, options]* Returns a new WriteStream object.
70	**fs.symlink***srcpath, dstpath[, type], callback* Asynchronous symlink. No arguments other than a possible exception are given to the completion callback. The type argument can be set to 'dir', 'file', or 'junction' $defaultis'file'$ and is only available on Windows $ignoredonotherplatforms$. Note that Windows junction points require the destination path to be absolute. When using 'junction', the destination argument will automatically be normalized to absolute path.