

🚩 Current Skill File System

Synchronous vs Asynchronous

File System

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")
```

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 an 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 latter does.

Example

Create a text file named **input.txt** with the following content:

```
Dummy Text For Testing !!!!
```



Let us create a .js file named **main.js** with 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: Dummy Text For Testing !!!!
```

```
Program Ended
```

```
Asynchronous read: Dummy Text For Testing !!!!
```

Let's take a look at a set of good examples on major File I/O methods:

Open a File

Syntax

The following is the method's syntax for opening a file in asynchronous mode:




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

Parameters

Here is the description of the parameters used :

- **path** – This is the string that has the file name and the 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).

Flags

- 
- **r** : Opens file for reading. An exception occurs if the file does not exist.
 - **r+** : Opens file for reading and writing. An exception occurs if the file does not exist.

- **rs** : Opens file for reading in synchronous mode.
- **rs+** : Opens file for reading and writing, asking the OS to open it synchronously. See notes for 'rs' about using this with caution.
- **w** : Opens 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+** : Opens 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** : Opens file for appending. The file is created if it does not exist.
- **ax** : Like 'a' but fails if the path exists.
- **a+** : Opens 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

Let us create a .js file named **main.js** that has the following code for opening an **input.txt** file 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!
```

Get a File's Information

Syntax

The following is one the methods' syntax for getting the file's information:

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

Parameters

Here is the description of the parameters used:

- **path** – This is the string that has the file name and the 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 several useful methods available in **fs.Stats** class which can be used to check file type. These methods are given in the following list:

- **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 a socket.

Example

Let us create a js file named **main.js** with 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 a File

Syntax

The following is one of the methods' syntax for writing 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

The following is one of the methods' syntax for reading from a file:



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

This method will use a 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 inside the file. If a 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.alloc(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
```

```
67 bytes read
```

```
Dummy Text For Testing!!!!
```

Closing a File

Syntax

The following is the syntax for closing 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
Dummy Text For Testing !!!!!
File closed successfully.

```

Truncate a File



Syntax

The following is the method's syntax for truncating 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** that has 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

Dummy Text

```
File closed successfully.
```

Delete a File



Syntax

The following is the method's syntax for deleting a file:

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

Parameters

Here is the description of the parameters used :

- **path** – This is the file name including its 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

The following is the method's syntax for creating a directory:

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

Parameters

Here is the description of the parameters used:

- **path** – This is the directory name including its 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 :

```
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

The following is the method's syntax for reading 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

The following is the method's syntax for removing 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 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
```

[< Previous](#)

[next >](#)

