Node.JS – Filesystem & Streams

Agenda

- Filesystem module overview
- Filesystem sync functions
- Filesystem async functions
- Buffers
- Streams module
- Filesystem streams

FILESYSTEM MODULE OVERVIEW

Filesystem Module overview

Sync Functions	Async Functions	Classes	Streams
• readFileSync • writeFileSync • mkdirSync • accessSync • statSync • …	• readFile • writeFile • mkdir • access • stat • fs.promises • …	• Dir • Dirent • FSWatcher • Stats • ReadStream • WriteStream	• createReadStream • createWriteStream • ReadStream • WriteStream

FILESYSTEM SYNC FUNCTIONS

FILESYSTEM – SYNC FUNCTIONS

- They all have the **Sync** suffix: readFileSync, writeFileSync, …
- They **will block** your code
- They are the easiest to use*
- Should not be used for code that is or can benefit from being async (like web services)
- Should not be used when the file sizes are unknown or known to be large.

readFileSync

- Path parameter can be a file path as string or buffer, it can be a URL and can be an integer representing a file descriptor
- Encoding by default is null, but if left undefined buffer is returned.
- Options can be string, in that case will be the encoding.

```
const fs = require('fs');

const fileContent = fs.readFileSync('./helloworld.txt', 'utf8');
console.log(fileContent); //Hello World!

```

```
1   const fs = require('fs');
2
3   const fileContent = fs.readFileSync('./helloworld.txt');
4
5   console.log(fileContent);
6   //<Buffer 48 65 6c 6c 6f 20 57 6f 72 6c 64 21>
7   console.log(fileContent.toString()); //Hello World!
8
```

writeFileSync

- File parameter works like readFileSync
- writeFileSync will replace the file if it already exists
- Encoding by default is utf8
- If data is buffer, encoding is ignored
- Options can be string, in that case it will be the encoding.

```
1   const fs = require('fs');
2
3   const fileData = 'lorem ipsum';
4
5   fs.writeFileSync('./my_new_file.txt', fileData, 'utf8');
6
7   console.log(
8       fs.readFileSync('./my_new_file.txt', 'utf8')
9   ); //lorem ipsum
```

```
1   const fs = require('fs');
2
3   const buffer = Buffer.from([
4       0x6c, 0x6f, 0x72, 0x65, 0x6d,
5       0x20, 0x69, 0x70, 0x73, 0x75, 0x6d
6   ]);
7
8   fs.writeFileSync('./from_buffer.txt', buffer);
9
10  console.log(
11      fs.readFileSync('./from_buffer.txt', 'utf8')
12  ); //lorem ipsum
13
```

mkdirSync

- Path parameter does not accept a file descriptor
- Recursive mode to create parent folders (Since node 10.12)
- Options can be an integer representing file mode on unix systems

```
1   const fs = require('fs');
2
3   fs.mkdirSync('./new_dir');
4
```

```
const fs = require('fs');

try {
    fs.mkdirSync('./parent/new_dir');
} catch (error) {
    console.error('Failed because directory parent doesn't exists');
}

//succeeds
fs.mkdirSync('./parent/new_dir', { recursive: true });

```

accessSync

- Test user’s permissions to file or directory specified by path.
- Mode option is useful for checking specific kinds of access, like: File_OK, Read Acess, Write Access, ...
- **Don’t** use access to check accessibility before opening/reading/writing. It is not recommended because introduces a race condition.

```javascript
const fs = require('fs');

const file = './index.js';

try {
    // Check if the file exists in the current directory.
    fs.accessSync(file, fs.constants.F_OK);
    console.log('file is ok');
} catch (error) {
    console.error('no access');
}

```

```
const fs = require('fs');

const file = './index.js';

try {
    // Check if the file is readable and writable.
    fs.accessSync(file, fs.constants.R_OK | fs.constants.W_OK);
    console.log('can read / write');
} catch (error) {
    console.error('no access');
}
```

statSync

- Get information about given file or directory
- Returns class fs.Stat, which contain very helpful methods and properties like: isDirectoy() isFIFO(), isFile() isSocket(), isSymbolicLink(),…
- **Don't** use stat to check accessibility before opening/reading/writing. It is not recommended because introduces a race condition.

```javascript
const fs = require('fs');

// Check if the file exists in the current directory.
const stats = fs.statSync('./index.js');

console.log(`Is symbolic link? ${stats.isSymbolicLink()}`);
console.log(`Is directory? ${stats.isDirectory()}`);
console.log(`Is file? ${stats.isFile()}`);
console.log(`created at: ${stats.birthtimeMs}`);

```

Error Handling in Sync methods

- So far, most of the examples didn’t had error handling. In real world code, you always need to handle errors in IO operations.
- All sync methods will throw on errors, so **try {} catch blocks are needed** for error handling.

```
1   const fs = require('fs');
2
3   try {
4       // will throw error because file does not exist
5       const fileContent = fs.readFileSync('./wrong_name.txt', 'utf8');
6       // ...
7   } catch (error) {
8       console.log(error.code); //ENOENT
9       console.error(`Could not read file ${error.path}`);
10  }
11
```

Note about Sync methods

- You should avoid using sync methods. As they block code, they can stop you entire microservice, webserver, program.
- Is ok to use for simple CLIs, utility scripts or programs that resemble a single-run tool and there is no intention of leveraging non-blocking execution.
- If you are in doubt, go for the async functions or stream API

FILESYSTEM ASYNC FUNCTIONS

FILESYSTEM – ASYNC FUNCTIONS

- They **don't** have a suffix: readFile, writeFile, access, stat …
- They **will not block** your code
- They cannot be stopped once started
- Functions like **readFile** will read the entire file to memory, so not the best option when the file size is unknown or known to be large.

Signature of async functions

- Signature of async functions are equal to the sync ones, except that they require a callback as last argument.
- On Node.js docs, most of the information resides in the async version.

```
1   const fs = require('fs');
2
3   fs.readFile('./helloworld.txt', 'utf8', (error, file) => {
4       console.log(file);
5   });
6   console.log('Reading file...');
7
8   //Reading file...
9   //Hello World!
10
```

```
1   const fs = require('fs');
2
3   const fileData = 'lorem ipsum';
4
5   fs.writeFile('./my_new_file.txt', fileData, 'utf8', (error) => {
6       console.log('file has been written');
7   });
8
```

Promise API

- The promise version of the functions don’t accept a callback as last parameter. Instead, they return a promise.
- They can be found at fs.promises
- They are a recent addition. On node 10 you will se an experimental warning if you use them.

```
1   const fs = require('fs');
2   const fsPromises = fs.promises;
3
4   fsPromises.readFile('./helloworld.txt', 'utf8')
5       .then((file) => console.log(file));
6
7   console.log('Reading file...');
8
9   //Reading file...
10  //Hello World!
11
```

```
1   const fs = require('fs');
2   const fsPromises = fs.promises;
3
4   async function read() {
5       const file = await fsPromises.readFile(
6           './helloworld.txt', 'utf8'
7       );
8       console.log('Reading file...');
9       console.log(file);
10  }
11  read();
12  //Reading file...
13  //Hello World!
```

Error handling in Async filesystem functions

- As said with the sync functions, you always must handle IO errors somehow.
- When using callbacks, the first parameter of the callback function will be an error object or null.
- When using promises, add a catch to the promise.
- When using await, add a try/catch block.

```
1   const fs = require('fs');
2
3   fs.readFile('./_some_file.txt', 'utf8', (error, file) => {
4       if (error) {
5           return console.error(error.message);
6       }
7       console.log(file);
8   });
9   console.log('Reading file...');
10
11  //Reading file...
12  //ENOENT: no such file or directory, open './_some_file.txt'
13
```

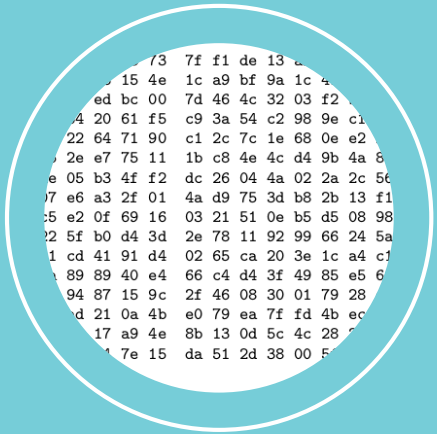
```
1   const fs = require('fs');
2   const fsPromises = fs.promises;
3
4   fsPromises.readFile('./_some_file.txt', 'utf8')
5       .then((file) => console.log(file))
6       .catch((error) => console.error(error.message));
7
8   console.log('Reading file...');
9
10  //Reading file...
11  //ENOENT: no such file or directory, open './_some_file.txt'
12
```

```
1   const fs = require('fs');
2   const fsPromises = fs.promises;
3
4   async function read() {
5       console.log('Reading file...');
6       try {
7           const file = await fsPromises.readFile(
8               './_some_file.txt', 'utf8'
9           );
10          console.log(file);
11      } catch (error) {
12          console.log(error.message);
13      }
14  }
15  read();
16  //Reading file...
17  //ENOENT: no such file or directory, open './_some_file.txt'
18
```

BUFFERS

BUFFERS

- API to manipulate binary data in node.
- Implements UInt8Array API
- Similar to arrays of integers from 0 to 255.
- Size of the buffer is stablished when it is created and cannot be changed.



Creating a Buffer

- **new Buffer()** is deprecated since node 6 because of security concerns.
- To create a buffer you can use Buffer.from(), Buffer.alloc(), and Buffer.allocUnsafe()
- allocUnsafe is faster and therefore might be required in performance critical paths of some applications. But it is, as the name imply, unsafe and a security risk.

```
1   //creates Buffer from string
2   const buff01 = Buffer.from('hello');
3   console.log(buff01);
4   //<Buffer 68 65 6c 6f>
5
6   //creates Buffer from string
7   const buff02 = Buffer.from([0x61, 0x62, 0x63]);
8   console.log(buff02);
9   //<Buffer 61 62 63>
10  console.log(buff02.toString());
11  //abc
```

```
//creates Buffer with 3 bytes filled with 0x1F
const buff01 = Buffer.alloc(3, 0x61);
console.log(buff01);
//<Buffer 61 61 61>
```

STREAMS

STREAMS

- Event based APIs for asynchronous data
- Can be piped together. (The output of one stream is the input of another)
- Can be paused of closed prematurely
- Can avoid memory overflow if used correctly
- Readable, Writable, Duplex and Transform streams

ReadableStream

- Are an abstraction from a source from which data is consumed. Examples: A file, STDIN, HTTP Request
- Can be consumed in two modes, **flowing** and **paused**
- In flowing mode, data is read **automatically** and provided as quickly as possible via the **EventEmitter** interface.
- In paused mode, the stream.read() method must be called explicitly to read chunks of data.

Readable stream flowing mode

- All Readable streams begin in paused mode but can be switched to flowing mode in one of the following ways
- Adding a 'data' event handler.
- Calling the stream.resume() method.
- Calling the stream.pipe() method to send the data to a Writable Stream.

```
1   const fs = require('fs');
2
3   //165MB of 'lore ipsum dolor'
4   const file = '../../resources/largeFile.txt';
5
6   const rStream = fs.createReadStream(file);
7
8   //in this case reading only starts
9   //when data eventListener is added
10  rStream.on('data', (chunk) => {
11      console.log(chunk.length); //64kb
12  });
```

```
6   const rStream = fs.createReadStream(file);
7
8   //in this case reading only starts
9   //when data eventListener is added
10  rStream.on('data', (chunk) => {
11      console.log(chunk.length); //64kb
12  });
13
14  rStream.on('close', () => {
15      console.log('file was closed');
16  });
17
18  rStream.on('error', (error) => {
19      console.error(error.message);
20  });
```

```
1   const fs = require('fs');
2
3   //165MB of 'lore ipsum dolor'
4   const file = '../../resources/largeFile.txt';
5
6   const rStream = fs.createReadStream(file, {
7       highWaterMark: 10 //make chunks max size 10 bytes
8   });
9
10  //It will take longer tan reading 64kb
11  //but uses less memory
12  rStream.on('data', (chunk) => {
13      console.log(chunk.length); //10 bytes
14  });
15
```

Readable stream flowing mode

- The important concept to remember is that a Readable will not generate data until a mechanism for either consuming or ignoring that data is provided. If the consuming mechanism is disabled or taken away, the Readable will attempt to stop generating the data.
- For backward compatibility reasons, removing 'data' event handlers will not automatically pause the stream. Also, if there are piped destinations, then calling stream.pause() will not guarantee that the stream will remain paused once those destinations drain and ask for more data.

The Readable stream API evolved across multiple Node.js versions and provides multiple methods of consuming stream data. In general, developers should choose one of the methods of consuming data and should never use multiple methods to consume data from a single stream. Specifically, using a combination of on('data'), on('readable'), pipe(), or async iterators could lead to unintuitive behavior.

Writable stream

- Writable streams are an abstraction for a *destination* to which data is written. Ex: HTTP Responses, STDOUT, File write streams.
- To write on them, call the write method or pipe it to a readable stream.

```
const fs = require('fs');

const writable = fs.createWriteStream('./file.txt', 'utf8');

writable.write('writing some data,');
writable.write(' writing some more data,');
writable.end(' last write,');
```

```javascript
const fs = require("fs");
const writable = fs.createWriteStream("./file.txt", "utf8");

function writeOneMillionTimes(writer, data, encoding, callback) {
  let i = 1000000;
  write();

  function write() {
    let ok = true;
    do {
      i--;
      if (i === 0) {
        // Last time!
        writer.write(data, encoding, callback);
      } else {
        // See if we should continue, or wait.
        // Don't pass the callback, because we're not done yet.
        ok = writer.write(data, encoding);
      }
    } while (i > 0 && ok);
    if (i > 0) {
      // Had to stop early!
      // Write some more once it drains.
      writer.once("drain", write);
    }
  }
}

writeOneMillionTimes(writable, "lorem ipsum", "utf8", error => {
    console.log('finished');
});
```

```
16        // See if we should continue, or wait.
17        // Don't pass the callback, because we're not done yet.
18        ok = writer.write(data, encoding);
```

```
20        } while (i > 0 && ok);
21        if (i > 0) {
22          // Had to stop early!
23          // Write some more once it drains.
24          writer.once("drain", write);
25        }
```

```
12        if (i === 0) {
13          // Last time!
14          writer.write(data, encoding, callback);
```

```
1   const zlib = require('zlib');
2   const fs = require('fs');
3
4   //165MB of 'lore ipsum dolor'
5   const file = '../../resources/largeFile.txt';
6
7   // streams
8   const gzip = zlib.createGzip(); //dulex stream
9   const readable = fs.createReadStream(file);
10  const writable = fs.createWriteStream('out.txt.gz');
11
12  readable.pipe(gzip).pipe(writable);
13  //out.txt.gz has 403Kb
14
```

```
1   const zlib = require('zlib');
2   const fs = require('fs');
3
4   //165MB of 'lore ipsum dolor'
5   const file = '../../resources/largeFile.txt';
6
7   //streams
8   const gzip = zlib.createGzip(); //dulex stream
9   const readable = fs.createReadStream(file);
10  const writable = fs.createWriteStream('out.txt.gz');
11
12  readable.pipe(gzip)
13      .on('error', () => { /* handle error */ })
14      .pipe(writable)
15      .on('error', () => { /* handle error */ });
16  //out.txt.gz has 403Kb
```

```
1   const zlib = require('zlib');
2   const fs = require('fs');
3   const { pipeline } = require('stream');
4
5   //165MB of 'lore ipsum dolor'
6   const file = '../../resources/largeFile.txt';
7
8   pipeline(
9       fs.createReadStream(file),
10      zlib.createGzip(),
11      fs.createWriteStream('out.txt.gz'),
12      (error) => {
13          if (error) { /* ... */ }
14          else { console.log('finished') }
15      }
16  );
17  //out.txt.gz has 403Kb
```

QUESTIONS?

EXTRA EXAMPLES/CONTENT

PATH MODULE

- Has several functions for working with paths, like path.join, path.resolve and path.relative
- Don’t ever try to manipulate paths by hand, always use the path module instead.
- Every module has a special variable called **__dirname** which contains the path for the module file itself.

```
1   const fs = require('fs');
2   const path = require('path');
3
4   //__dirname is a special variable with the path of the module
5   //join a couple of .. to navigate back two directories
6   //don't try to concatenate those string by hand, always use path.join
7   const filepath = path.join(__dirname, '../..', 'resources/helloworld.txt');
8
9   const fileContent = fs.readFileSync(filepath, 'utf8');
10
11  console.log(fileContent); //Hello World!
12
```

```
1   const fs = require('fs');
2   const path = require('path');
3
4   let fileContent;
5   try {
6       //using relative paths directly is dangerous
7       //because it is relative to the CWD and not to this module.
8       fileContent = fs.readFileSync('./helloworld.txt', 'utf8');
9   } catch (e) {
10      console.error('Error reading file, use path module');
11  }
12
13  //__dirname is a special variable with the path of the module
14  const filepath = path.resolve(__dirname, './helloworld.txt');
15
16  console.log(filepath);
17  fileContent = fs.readFileSync(filepath, 'utf8');
18
19  console.log(fileContent); //Hello World!
20
```

Custom readablestream class

The next example shows the custom readable stream that was used to make a file with 10 million lines for the other examples.

```
1  const { Readable } = require('stream');
2  const fs = require('fs');
3
4  class loreIpsumDolor extends Readable {
5    constructor(totalIterations) {
6      super({ highWaterMark: 500 }); //only read chunks of 500 bytes
7      this.totalIterations = totalIterations;
8      this.currentIterations = 0;
9    }
10
11   _read() { //we cannot override the "read" method, instead we override the _read
12     this.currentIterations++;
13
14     if (this.currentIterations > this.totalIterations) {
15       //push null to end the reading
16       this.push(null);
17       return;
18     }
19
20     this.push('lore ipsum dolor\n');
21   }
22 }
23
24 const writeStream = fs.createWriteStream('../../resources/largeFile.txt', 'utf8');
25 new loreIpsumDolor(10000000).pipe(writeStream); //Write 10 million times to file
26
```

Example on how to read a file line-by-line

The next example shows the usage of the module called readline. In this example it was used to read a file line-by-line

```
1   const fs = require('fs');
2   const readline = require('readline'); //built-in node module
3
4   const rl = readline.createInterface({
5     input: fs.createReadStream('./file.txt'), //read stream
6     crlfDelay: Infinity //make sure we read \r\n correctly
7   });
8
9   rl.on('line', (line) => { //get line instead of chunk
10    console.log(`Line from file: ${line}`);
11  });
12
```