NODE.JS GLOBAL MENOTING

FILESYSTEM AND STREAMS

January, 2019

AGENDA

- FS module
- Working with files
- Streams
- Readable and writable streams
- Child process module
- Cluster module

FS MODULE

Operates with classes:

- Stats – describe base information about file or directory
- FSWatcher – allow us to work with file changes
- ReadStream – readable file Stream
- WriteStream – writable file Stream
- FileHandle – wrapper for File Descriptor

Also uses constants and flags:

- Access (work with fs.access())
- Copy (work with fs.copyFile())
- File open (work with fs.open() that also can be modified by flags, e.g. “r” or “w”)
- File Type and Mode (work with Stats class)

Methods groups:

- File content
- Placement
- Directories
- Properties and permissions
- File Descriptor lifecycle
- Events handling
- Streams

Feature of Node.JS 10:

- fsPromises module
- FileHandle – File Descriptor for fsPromises

FS MODULE METHODS

File content:

- readFile*^
- writeFile*^
- appendFile*^
- read*^
- write*^
- ftruncate*^/truncate*^
- fdatasync*^/fsync*^

Placement:

- copyFile*
- rename*
- link*
- symlink*
- readlink*
- realpath*
- unlink*

Directories:

- mkdir*
- mkdtemp*
- readdir*
- rmdir*

Properties and Permissions:

- access*
- exists
- fstat*/stat*^/lstat*
- futimes*/utimes*^
- chmod*^/fchmod*/lchmod*
- chown*^/fchown*/lchown*

File Descriptor:

- open*
- close^

Handling:

- watch
- watchFile
- unwatchFile

Streams:

- createReadStream
- createWriteStream

* - **fsPromises** submodule;
^ - **fileHandle** has own

WORKING WITH FILES (BASIC EXAMPLE)

```
1  /* short_text.txt -> I could either watch it happen or be a part of it. */
2
3  const fs = require('fs');
4  const data = fs.readFileSync('./data/short_text.txt');
5
6  console.log(`Content -> `, data);
7  console.log(`Content class -> ${data.constructor.name}`);
8  console.log(`Converted content -> ${data.toString()}`);
```

```
➜ training examples node ex1.js
Content ->  <Buffer 49 20 63 6f 75 6c 64 20 65 69 74 68 65 72 20 77 61 74 63 68 20 69
 74 20 68 61 70 70 65 6e 20 6f 72 20 62 65 20 61 20 70 61 72 74 20 6f 66 20 69 74 2e>
Content class -> Buffer
Converted content -> I could either watch it happen or be a part of it.
```

CLASS BUFFER AND BINARY FILESYSTEM

How our text looks like in HEX format:

```
00 01 02 03 04 05 06 07 08 09 0a 0b 0c 0d 0e 0f
49 20 63 6f 75 6c 64 20 65 69 74 68 65 72 20 77   I could either w
61 74 63 68 20 69 74 20 68 61 70 70 65 6e 20 6f   atch it happen o
72 20 62 65 20 61 20 70 61 72 74 20 6f 66 20 69   r be a part of i
74 2e                                             t.
```

Data in **binary** format	0100 1001	0010 0000	0110 0011
Data in **byte** format	49	20	63
Data in **char** format	I	*whitespace*	c

CLASS BUFFER

Buffer – class which wraps ”Uint8Array” typed array of 8-bit unsigned integers. (dec: 0 – 255; hex: 0 – FF; binary: 0000 0000 – 1111 1111)

Size of the buffer is determined at initialization and after that it cannot be changed.

Using Buffer as constructor “new Buffer” is deprecated, use instead:

- Buffer.alloc()
- Buffer.concat()
- Buffer.from()

Methods:

- toString() - decodes to a string according of the specified character encoding (utf-8 by default)

Properties:

- [index] – index iterator (inherited from Uint8Array)
- length – size of memory given for current buffer

FILE DESCRIPTOR

Let imagine that we need to read first 1/5 part of file:

```
1   const fs = require('fs');
2   const filePath = './data/short_text.txt';
3
4   const fileDescriptor = fs.openSync(filePath, 'r');
5   const fileInfo = fs.fstatSync(fileDescriptor);
6   const buffer = Buffer.alloc(fileInfo.size);
7
8   const bufferStartOffset = 0;
9   const fileStartPosition = 0;
10  const length = fileInfo.size / 5;   // 50/5 = 10 symbols should be shown
11
12  const bytesRead = fs.readSync(fileDescriptor, buffer, bufferStartOffset, length, fileStartPosition)
13  console.log(`File with Descriptor ${fileDescriptor} (bytes read: ${bytesRead}): ${buffer}`);
14
15  fs.closeSync(fileDescriptor);
```

```
➜ training examples node ex2.js
File with Descriptor 13 (bytes read: 10): I could ei
```

FileDescriptor – integer value which represents reference at an open file.
FileHandle – wrapper on File Descriptor for fsPromises submodule.

WATCHING FILE CHANGES

Two variants:

- watchFile / unwatchFile()
- watch() – newer implementation of watchFile/unwatchFile and should be always used instead of them. Return an instance of FSWatcher class.

Class FSWatcher:

- event ‘change’ – emits when file changed, callback gets two arguments (eventType [‘rename’ or ’change’, filename)
- event ‘close’ – emits when stops watching for changes
- event ‘error – emits when an error occurs
- method ‘close()’ – for stop watching the file

WATCHIN FILE CHANGES (EXAMPLE)

```
1  const fs = require('fs');
2  const filePath = './data/writeHere.txt';
3
4  const watcher = fs.watch(filePath);
5
6  const appendData = (data) => {
7      console.log('append   - ', new Date(), `data - ${data}`);
8      fs.appendFileSync(filePath, data);
9  }
10
11 const stopWatch = () => {
12     console.log('stop     - ', new Date());
13     watcher.close();
14 }
15
16 console.log('start    > ', new Date());
17 appendData('Elon ');
18
19 watcher.on('change', (eventType, fileName) => {
20     console.log('changed  - ', new Date(),`
21     Event type: ${eventType}; File Name: ${fileName}`);
22 });
23
24 setTimeout(() => { appendData("Musk ") }, 5000);
25 setTimeout(() => { appendData("launches rockets in space."); stopWatch() }, 10000);
```

```
➜ training examples node ex3.js
start    >  2019-01-13T16:19:35.068Z
append   -  2019-01-13T16:19:35.074Z data - Elon
changed  -  2019-01-13T16:19:35.509Z
    Event type: change; File Name: writeHere.txt
append   -  2019-01-13T16:19:40.083Z data - Musk
changed  -  2019-01-13T16:19:40.084Z
    Event type: change; File Name: writeHere.txt
append   -  2019-01-13T16:19:45.080Z data - launches rockets in space.
stop     -  2019-01-13T16:19:45.081Z
➜ training examples
```

CLASS STREAM

What:

- Abstract interface for working with streaming data in Node.js

Why:

- To easily build objects that implement the stream interface and represent flowing data from any source

Types:

- Readable – streams from which data can be read, e.g. fs.createReadStream
- Writable – streams to which data can be written, e.g. fs.createWriteStream
- Duplex – streams that are both Readable and Witable, e.g. net.Socket
- Transform – streams that can modify or transform the data, e.g. zlib.createDeflate

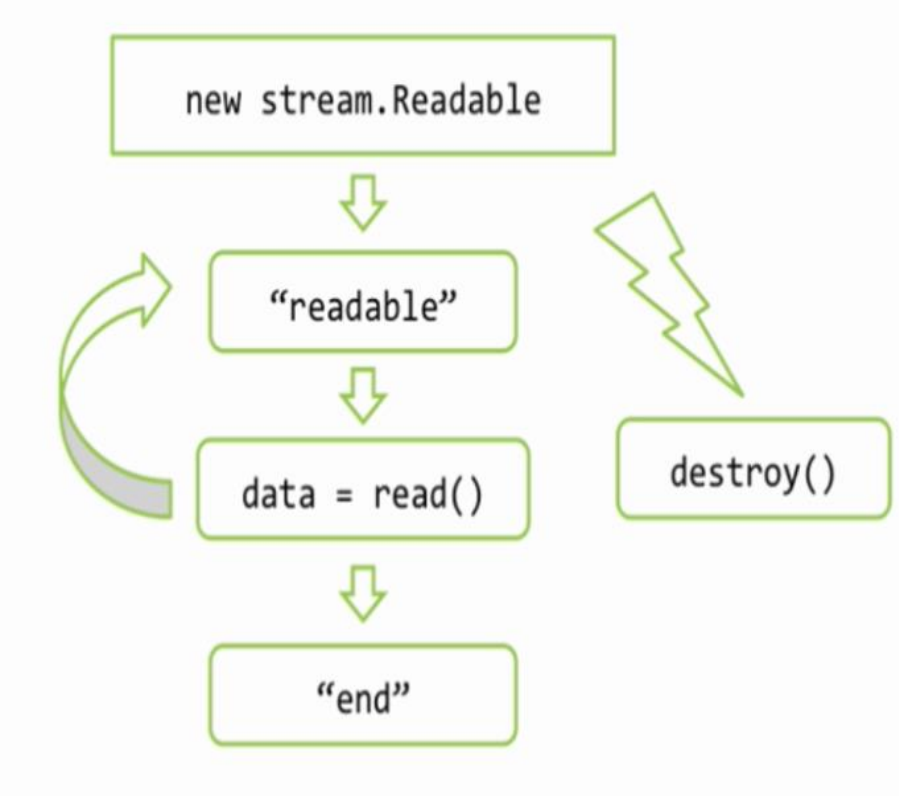
READABLE STREAM

Events:

- ‘readable’ – when stream ready for reading from his internal buffer
- ‘error’ – emits when an error occurs
- ‘end’ – when achieved end of source data

Methods

- read() – read chunk of data from internal buffer of stream
- read(N) – read chunk of data with N bytes



READ FILE STREAM IN PAUSE MODE EXAMPLE

```
const fs = require('fs');
const filePath = './data/huge_amount_of_Elon_Musk.txt';

const reader = fs.createReadStream(filePath);

let emits = 0;
let chunks = 0;
let symbols = 0;

function logStatus() {…
}

function read() {
    let chunk = null;
    while (null !== (chunk = reader.read())) {
        chunks++;
        symbols += chunk.toString().length;
    }
}

reader.on('readable', () => {
    emits++;
    read();
    if (emits < 3) { logStatus(); }
});

reader.on('end', () => { logStatus(); console.log('Stream ended.')})
```

File with 100k phrases “Elon Musk launches rockets in space with tesla MS.” (50 sym length)

```
➜ training examples node ex4.js
Emits:  1
Chunks:  1
Symbols:  65536

Emits:  2
Chunks:  2
Symbols:  131072

Emits:  764
Chunks:  763
Symbols:  50000000

Stream ended.
➜ training examples
```

READ FILE STREAM IN PAUSE MODE EXAMPLE

```
1   const fs = require('fs');
2   const filePath = './data/huge_amount_of_Elon_Musk.txt';
3
4   const reader = fs.createReadStream(filePath);
5
6   let emits = 0;
7   let chunks = 0;
8   let symbols = 0;
9
10  function logStatus() {…
14  }
15
16  function read() {
17      let chunk = null;
18      while (null !== (chunk = reader.read(25))) {
19          chunks++;
20          symbols += chunk.toString().length;
21      }
22  }
23
24  reader.on('readable', () => {
25      emits++;
26      read();
27      if (emits < 3) { logStatus(); }
28  });
29
30  reader.on('end', () => { logStatus(); console.log('Stream ended.')})
```

File with 100k phrases “Elon Musk launches rockets in space with tesla MS.” (50 sym length)

```
➜ training examples node ex4.js
Emits:  1
Chunks:  2621
Symbols:  65525

Emits:  2
Chunks:  5242
Symbols:  131050

Emits:  764
Chunks:  2000000
Symbols:  50000000

Stream ended.
➜ training examples
```

READABLE STREAM MODES:

Pause (by default):

- By calling the stream.pause() mehod
- By removing any ‘data’ event handlers and all pipe destinations by calling the stream.unpipe() method

Flow:

- Adding a ‘data’ event handler
- Calling the stream.resume() method
- Calling the stream.pipe() method to send the data to a Writable

READ FILE STREAM IN FLOW MODE EXAMPLE

```
1   const fs = require('fs');
2   const filePath = './data/huge_amount_of_Elon_Musk.txt';
3   const reader = fs.createReadStream(filePath);
4
5   let emits = 0;
6   let symbols = 0;
7
8   function logStatus() {…
11  }
12
13  reader.on('data', (chunk) => {
14      emits++;
15      symbols += chunk.toString().length;
16      if (emits < 3) { logStatus(); }
17  });
18
19  reader.on('end', () => {
20      logStatus();
21      console.log('Stream ended.');
22  });
```

```
➜ training examples node ex5.js
Emits:  1
Symbols:  65536

Emits:  2
Symbols:  131072

Emits:  763
Symbols:  50000000

Stream ended.
➜ training examples
```

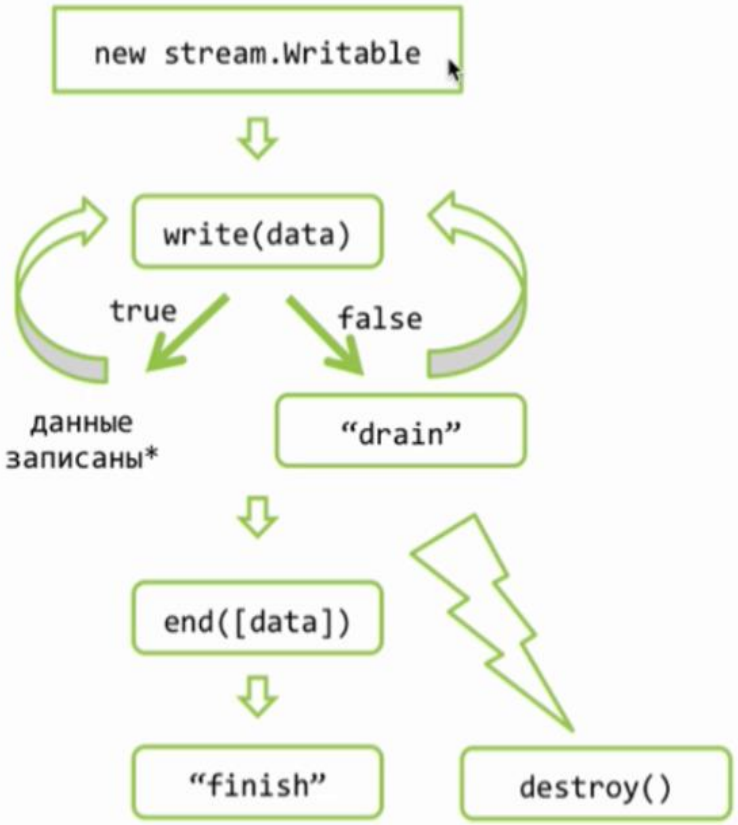
WRITABLE STREAM BASIC

Events

- 'drain' – when internal buffer is ready to get new data by write() method
- 'error' – emits when an error occurs
- 'finish' - when end() called after all data from internal buffer where written
- 'pipe/unpipe' – when stream is piped/unpiped to readable stream

Methods:

- write() – write data to internal buffer of stream, return flag
- end() – finish working with stream, can take last chunk of data to write



WRITE FILE STREAM EXAMPLE

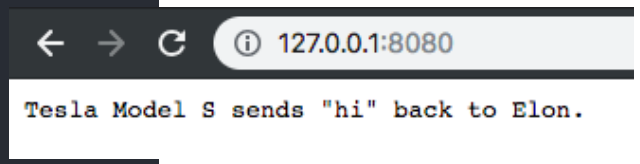
```
const fs = require('fs');
const filePath = './data/huge_amount_of_Elon_Musk1.txt';

const writer = fs.createWriteStream(filePath);
const sentence = 'Elon Musk launch rockets in space and tesla MS.' // 50 symbols
const count = 1E+6;

let index = 0;
let drainCounter = 0;

function write() {
    if (index % 1E+5 === 0) {
        console.log(`Step: ${index / 1E+5}; Drain counter: ${drainCounter};`);
    }

    if (index < count) {
        index++;
        if (writer.write(sentence)) {
            write();
        } else {
            writer.once('drain', () => {
                drainCounter++;
                write();
            });
        }
    } else {
        writer.end();
    }
}

writer.on('finish', () => console.log(`Finished with ${drainCounter} drains.`));
write();
```

```
➜ training examples node ex6.js
Step: 0; Drain counter: 0;
Step: 1; Drain counter: 286;
Step: 2; Drain counter: 573;
Step: 3; Drain counter: 859;
Step: 4; Drain counter: 1146;
Step: 5; Drain counter: 1432;
Step: 6; Drain counter: 1719;
Step: 7; Drain counter: 2005;
Step: 8; Drain counter: 2292;
Step: 9; Drain counter: 2578;
Step: 10; Drain counter: 2865;
Finished with 2865 drains.
➜ training examples
```

REQUEST/RESPONSE

```
1   const http = require('http');
2   const port = 8080;
3   const server = http.createServer();
4
5   server.on('request', function(request, response) {
6       response.writeHead(200);
7       console.log(`${request.method}: ${request.url}`);
8       response.write('Tesla Model S sends "hi" back to Elon.')
9       response.end();
10  })
11
12  server.listen(port);
13  console.log(`Browse to http://127.0.0.1:` + port);
```

```
➜ training examples node ex7.js
Browse to http://127.0.0.1:8080
GET: /
```



Request:

- Instance of class **http.IncomingMessage**
- Implements **Stream** with type Readable

Response:

- Instance of class **http.ServerResponse**
- Implements **Stream** with type Readable

PIPE/UNPIPE METHOD

- Connects writable stream to readable stream
- Automatically transfers read data to writable stream
- Automatically manages things like handling errors, end-of-files, cases when one stream is slower or faster than the other
- Not recommended to mix with manual event handling (always choose: handle events or create pipe)

Pipe method:

```
const fs = require('fs');
const fromFile = './data/huge_amount_of_Elon_Musk.txt';
const toFile = './data/huge_huge_amount_of_Elon_Musk.txt';

const reader = fs.createReadStream(fromFile);
const writer = fs.createWriteStream(toFile);

reader.pipe(writer);
```

Can be simplified

Simplified event-equivalent method:

```
const fs = require('fs');
const fromFile = './data/huge_amount_of_Elon_Musk.txt';
const toFile = './data/huge_huge_amount_of_Elon_Musk.txt';

const reader = fs.createReadStream(fromFile);
const writer = fs.createWriteStream(toFile);

reader.on('data', (chunk) => {
    writer.write(chunk);
});

reader.on('end', () => { writer.end() });
```

```
reader.pipe(patcher).pipe(ecryptor).pipe(packer).pipe(writer);
```

CHILD PROCESS MODULE

‘child_process’ module for creating new process on OS and managing them.

Methods (all returns ChildProcess instance):

- exec(command[, options][, callback]) – spawn a subshell and execute the command in that shell
- execFile(file[, args][, options][, callback]) – executes an external application
- fork(modulePath[, args][, options]) - spawn new Node.JS instance with running module in it
- spawn(command[, args][, options]) – spawn an external application in a new process and returns a stream for I/O
- -//- synchronous analogs

Class ChildProcess:

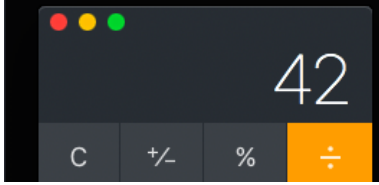
- Emits child process events (close, disconnect, error, exit, message)
- Lets send signals to child process (send, disconnect, kill)
- Contains readable and writable streams for transferring data

CHILD PROCESS MODULE USAGE EXAMPLE

```
const cp = require('child_process');
const modulePath = './ex10-2.js';
const child = cp.fork(modulePath);

setTimeout(() => {
    child.send('open /Applications/Calculator.app');
}, 1000);

child.on('message', (msg) => { console.log(`<P> Child reported: ${msg}`); });
child.on('close', () => { console.log('<P> Child exited.'); });
```

```
const cp = require('child_process');

process.on('message', (msg) => {
    console.log('<C> Got message from Parent: ', msg);

    const child = cp.exec(msg, (err) => {
        process.send(!err ? 'success' : 'fail');
        process.exit();
    });
});
```

```
➜ training examples node ex10-1.js

<C> Got message from Parent:  open
/Applications/Calculator.app
<P> Child reported: success
<P> Child exited.
➜ training examples
```



CLUSTER MODULE

“cluster” – module for horizontal scaling Node.JS application

Method:

- fork() – spawn a new worker process, returns Worker instance

Properties:

- isMaster/isWorker – is/isn’t current process master-process
- worker – reference to the current worker object (not available in the master process)
- workers – object with IDs as keys and workers as values (only available in maser process)

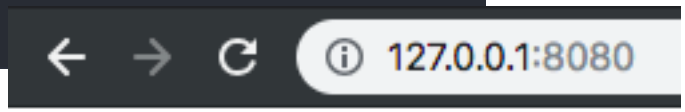
Class Worker:

- wraps ChildProcess instance which was originally created by *child_process.fork()*
- property “id” – unique id for worker, key in *cluster.workers*
- property “process” – ChildProcess instance
- method “send()” – send a message to master (from worker) or to worker (from master)

CLUSTER MODULE USAGE EXAMPLE

```
1   const cluster = require('cluster');
2   const http = require('http');
3   const numCPUs = require('os').cpus().length;
4
5   if (cluster.isMaster) {
6       console.log(`Master ${process.pid} is running`);
7
8       for (let i = 0; i < numCPUs; i++) {
9           cluster.fork();
10      }
11  } else {
12      http.createServer((req, res) => {
13          res.writeHead(200);
14          res.end(`Hi! My ID: ${cluster.worker.id} and PID: ${process.pid}.`)
15      }).listen(8080);
16
17      console.log(`Worker ${process.pid} started.`);
18  }
```

```
➜ training examples node ex11.js
Master 62529 is running
Worker 62530 started.
Worker 62532 started.
Worker 62531 started.
Worker 62533 started.
```



Hi! My ID: 1 and PID: 62530.

Cluster automatically recognizes which worker isn't under high load

USEFUL LINKS

NODE.JS GLOBAL

NODE.JS FILESYSTEM AND STREAMS

BY
GENNADII MISHCHENKO
DENIS VLASSENKO
MIKITA SUBBOT