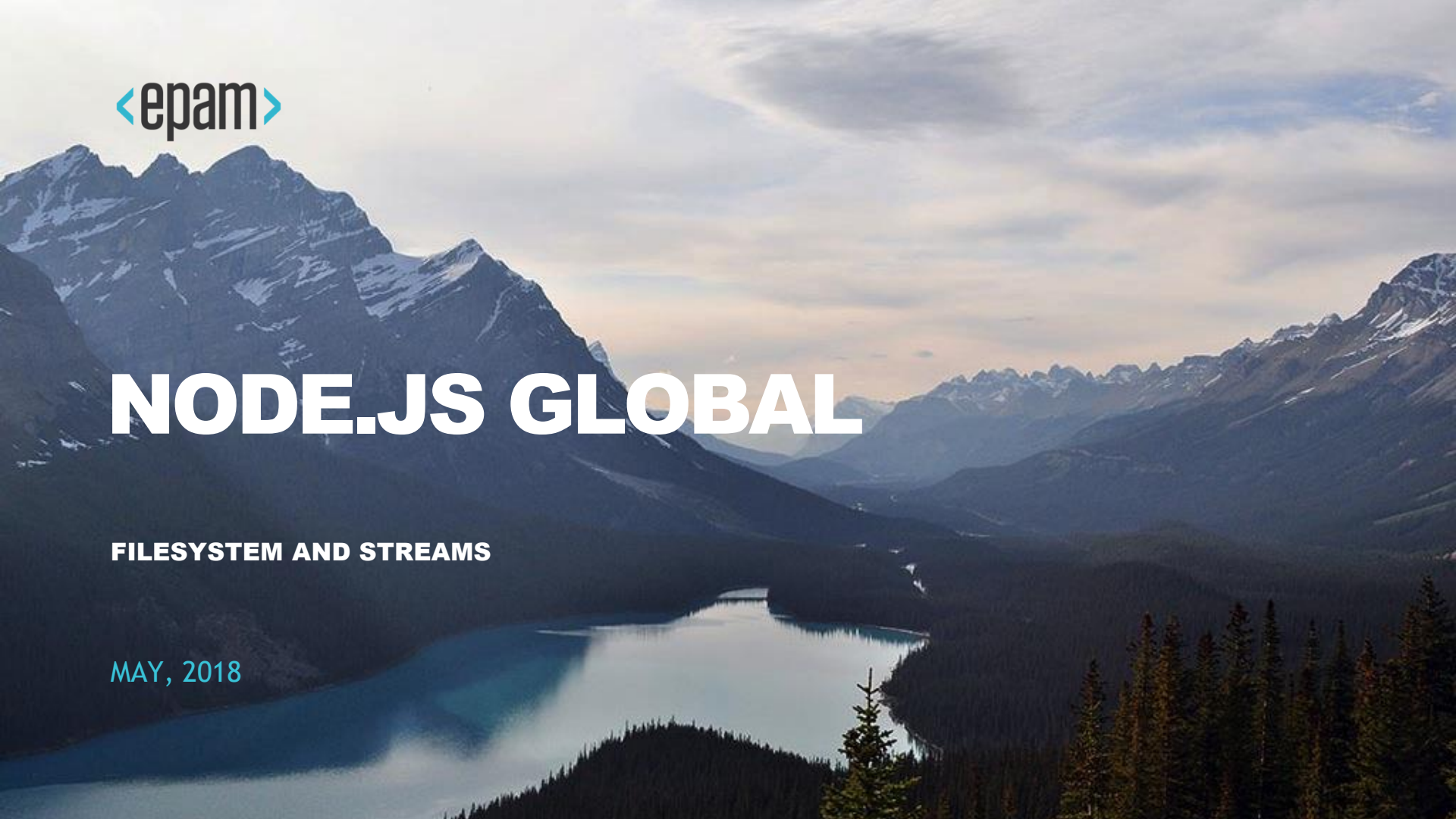


NODE.JS GLOBAL

FILESYSTEM AND STREAMS

MAY, 2018

AGENDA

- FS module
- Working with files
- Streams
- Readable/Writable streams
- Child process module
- Cluster module

FS MODULE

Methods (can be divided into groups):

- file content
- placement
- directories
- properties and permissions
- file descriptor lifecycle
- events handling
- streams

Classes:

- **Stats** – contains information about file or directory
- **FSWatcher** – lets us to handle file changes
- **ReadStream** – readable file Stream
- **WriteStream** – writable file Stream
- **FileHandle*** - wrapper for File Descriptor

Constants and Flags

submodule ‘promises’ (NodeJS v10):

- **fsPromises** – collection of promisified methods
- **FileHandle** – class that wraps **FileDescriptor** for **fsPromises** methods

FS MODULE METHODS

File content:

- readFile*^
- writeFile*^
- appendFile*^
- read*^
- write*^
- ftruncate*/truncate*^
- fdatasync*/fsync*
- ^(datasync/sync)

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

* - **fsPromise** also contains

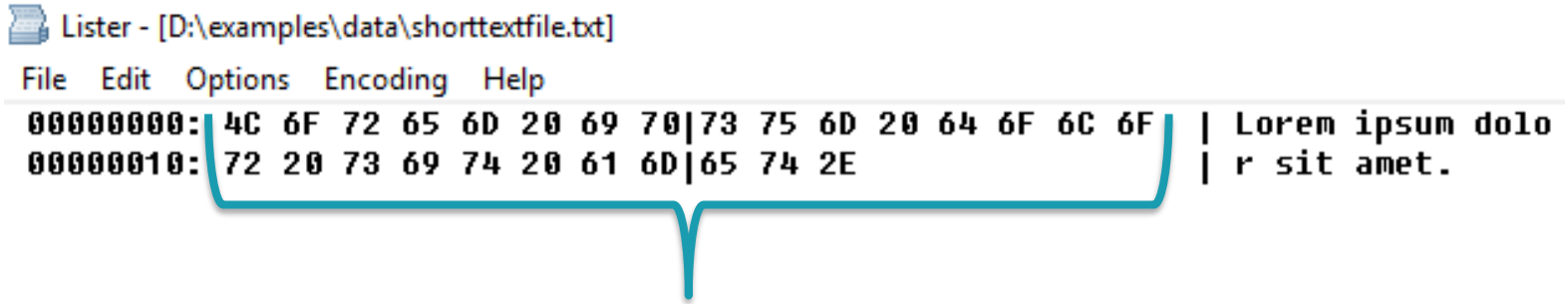
^ - **filehandle** has own

WORKING WITH FILES (BASIC EXAMPLE)

```
1  /* shorttextfile.txt = 'Lorem ipsum dolor sit amet.' */
2
3  const fs = require('fs');
4  const data = fs.readFileSync('./data/shorttextfile.txt');
5
6  console.log('- content:', data);
7  console.log('- content class:', data.constructor.name);
8  console.log('- converted content:', data.toString());
```

```
D:\examples>node 01.js
- content: <Buffer 4c 6f 72 65 6d 20 69 70 73 75 6d 20 64 6f 6c 6f 72 20 73 69 74 20 61 6d 65 74 2e 0d 0a>
- content class: Buffer
- converted content: Lorem ipsum dolor sit amet.
```

CLASS BUFFER AND BINARY FILESYSTEM



data stores in binary format	0100 1100	0110 1111	0111 0010
we work with it in byte format	4C	6F	72
in text file each char stores as byte code	L	o	r

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)

The size of the Buffer is established when it is created and cannot be resized.

Constructor **“new Buffer()” is deprecated**, use:

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

Methods:

- toString() - decodes to a string according to the specified character encoding (default: utf8)

Properties:

- [index] – index iterator, which is inherited from Uint8Array
- length - amount of memory allocated for current buffer in bytes

FILE DESCRIPTOR

```
1  const fs = require('fs');
2  const filepath = './data/shorttextfile.txt';
3
4  const fd = fs.openSync(filepath, 'r');
5  const fileInfo = fs.fstatSync(fd);
6  const buffer = Buffer.alloc(fileInfo.size);
7  const bufferStartOffset = 0;
8  const length = fileInfo.size / 3; // 9 = 27 / 3
9  const fileStartPosition = 0;
10
11 const bytesRead = fs.readSync(fd, buffer, bufferStartOffset, length, fileStartPosition);
12 console.log(`- content of <${fd}> (bytes read: ${bytesRead}): ${buffer}`);
13
14 fs.closeSync(fd);
```

```
D:\examples>node 02.js
- content of <3> (bytes read: 9): Lorem ips
```

File Descriptor (fd) – is integer value, which represents reference to an open file

FileHandle – wrapper on File Descriptor for fsPromise methods

WATCHING FILE CHANGES

Two variants:

- watchFile() / unwatchFile()
- watch() - Is newer and more efficient and should be used instead watchFile() / unwatchFile() when possible. Returns instance of class FSWatcher.

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

WATCHING FILE CHANGES (EXAMPLE)

```
1  const fs = require('fs');
2  const fp = './data/shorttextfile.txt';
3  const watcher = fs.watch(fp);
4  function append(data) {
5      console.log('append  -', new Date());
6      fs.appendFileSync(fp, data);
7  }
8  function stopWatch() {
9      console.log('stop    <', new Date());
10     watcher.close();
11 }
12
13 console.log('start   >', new Date());
14 append(1);
15
16 watcher.on('change', (eventType, filename) => {
17     console.log('changed -', new Date(), `
18     e: ${eventType}; fn: ${filename}`);
19 });
20
21 setTimeout(() => { append(2); }, 5000);
22 setTimeout(() => { append(3); stopWatch(); }, 10000);
```

```
D:\examples>node 03.js
start   > 2017-09-29T10:46:26.398Z
append  - 2017-09-29T10:46:26.400Z
changed - 2017-09-29T10:46:26.403Z
    e: change; fn: shorttextfile.txt
append  - 2017-09-29T10:46:31.403Z
changed - 2017-09-29T10:46:31.405Z
    e: change; fn: shorttextfile.txt
append  - 2017-09-29T10:46:36.402Z
stop    < 2017-09-29T10:46:36.403Z
```

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 (ex.: fs.createReadStream, process.stdin)
- Writable - streams to which data can be written (ex.: fs.createWriteStream, process.stdout)
- Duplex - streams that are both Readable and Writable (ex.: net.Socket)
- Transform - streams that can modify or transform the data (ex.: zlib.createDeflate)

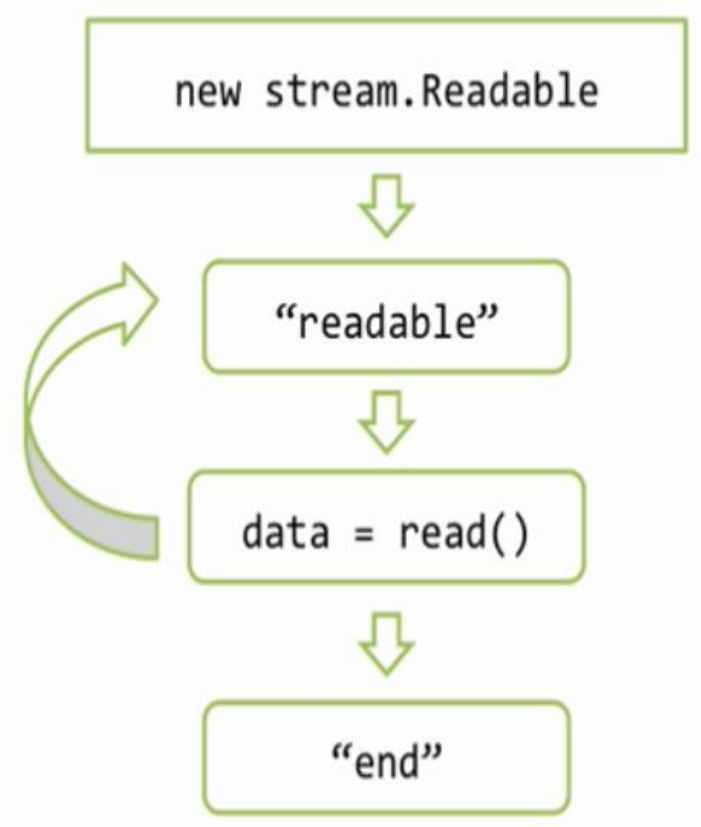
READABLE STREAM

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 size N bytes



READ FILE STREAM IN PAUSE MODE EXAMPLE

```
1  const fs = require('fs');
2  /* bigtextfile.txt = 1E+6 x 'Lorem ipsum dolor sit amet.' */
3  const filePath = './data/bigtextfile.txt';
4  const reader = fs.createReadStream(filePath);
5  let emits = 0;
6  let chunks = 0;
7  let symbols = 0;
8  function statusPrint() {···
12 }
13
14 function read() {
15     let chunk = null;
16     while (null !== (chunk = reader.read())) {
17         chunks++;
18         symbols += chunk.toString().length;
19     }
20 }
21
22 reader.on('readable', () => {
23     emits++;
24     read();
25     if (emits < 3) { statusPrint(); }
26 });
27 reader.on('end', () => { statusPrint(); console.log('Finished.'); });
```

```
D:\examples>node 04.js
emits: 1
chunks: 1
symbols: 65536

emits: 2
chunks: 2
symbols: 131072

emits: 413
chunks: 412
symbols: 27000000

Finished.
```

READ FILE STREAM IN PAUSE MODE EXAMPLE

```
    let chunk = null;
    while (null !== (chunk = reader.read(27))) {
        chunks++;
    }
```

```
D:\examples>node 04.js
emits: 1
chunks: 2427
symbols: 65529

emits: 2
chunks: 4854
symbols: 131058

emits: 413
chunks: 1000000
symbols: 27000000

Finished.
```

READABLE STREAM MODES

Pause (default):

- By calling the stream.pause() method
- 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/bigtextfile.txt';
3  const reader = fs.createReadStream(filePath);
4  let emits = 0;
5  let symbols = 0;
6  function statusPrint() {···
9  }
10
11 reader.on('data', (chunk) => {
12     emits++;
13     symbols += chunk.toString().length;
14     if (emits < 3) { statusPrint(); }
15 });
16
17 reader.on('end', () => {
18     statusPrint();
19     console.log('Finished.');
20 });
```

```
D:\examples>node 05.js
emits: 1
symbols: 65536

emits: 2
symbols: 131072

emits: 412
symbols: 27000000

Finished.
```

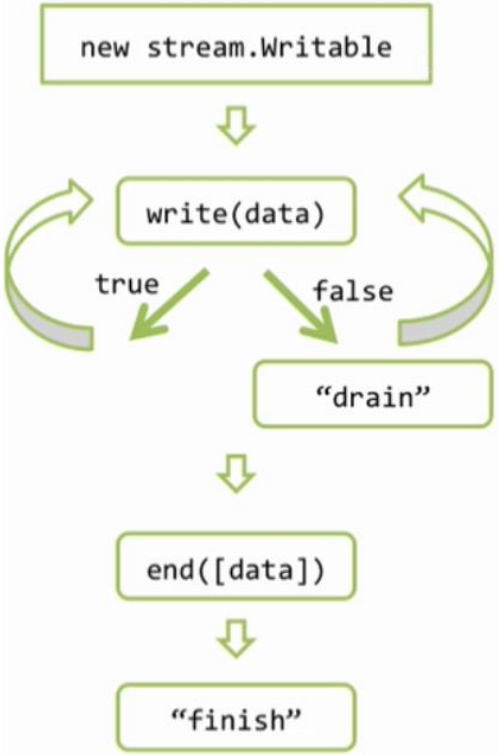
WRITABLE STREAM BASIS

Events:

- ‘drain’ – when internal buffer is ready to get new data by write() method
- ‘error’ – emits when an error occurs
- ‘finish’ – when end() called and after all data from internal buffer were 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

```
1  const fs = require('fs');
2  const filePath = './data/bigtextfile.txt';
3  const writer = fs.createWriteStream(filePath);
4  const sentence = 'Lorem ipsum dolor sit amet.'; // 27 symbols
5  const count = 1E+6;
6  let index = 0;
7  let drainCounter = 0;
8
9  function write() {
10     if (index % 1E+5 === 0)
11         console.log(`step: ${index / 1E+5}; drainCounter: ${drainCounter}`);
12
13     if (index < count) {
14         index++;
15         if (writer.write(sentence)) {
16             write();
17         } else {
18             writer.once('drain', () => { drainCounter++; write(); })
19         }
20     } else {writer.end();}
21 }
22 writer.on('finish', () => { console.log(`Finished with ${drainCounter} drains.`); });
23 write();
```

```
D:\examples>node 06.js
step: 0; drainCounter: 0
step: 1; drainCounter: 164
step: 2; drainCounter: 329
step: 3; drainCounter: 494
step: 4; drainCounter: 658
step: 5; drainCounter: 823
step: 6; drainCounter: 988
step: 7; drainCounter: 1153
step: 8; drainCounter: 1317
step: 9; drainCounter: 1482
step: 10; drainCounter: 1647
Finished with 1647 drains.
```

REQUEST/RESPONSE

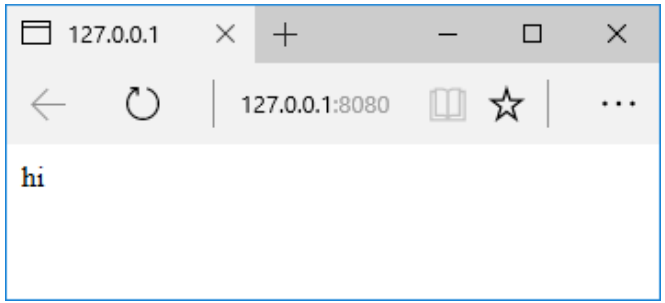
Request:

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

Response:

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

```
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('hi');
9      response.end();
10 });
11
12 server.listen(port);
13 console.log('Browse to http://127.0.0.1:' + port);
```




```
D:\examples>node 07.js
Browse to http://127.0.0.1:8080
GET: /
```

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)

```
1  const fs = require('fs');
2  const fromFile = './data/bigtextfile.txt';
3  const toFile = './data/bigtextfile_copy.txt'
4  const reader = fs.createReadStream(fromFile);
5  const writer = fs.createWriteStream(toFile);
6
7  reader.pipe(writer);
```

SIMPLIFIED event-equivalent code

```
1  const fs = require('fs');
2  const fromFile = './data/bigtextfile.txt';
3  const toFile = './data/bigtextfile_copy.txt'
4  const reader = fs.createReadStream(fromFile);
5  const writer = fs.createWriteStream(toFile);
6
7  reader.on('data', (chunk) => {
8      writer.write(chunk);
9  });
10
11 reader.on('end', () => { writer.end(); });
```

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

CHILD PROCESS MODULE

«child_process» - module for creating new processes in 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]) – spawns an external application in a new process and returns a streaming interface 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

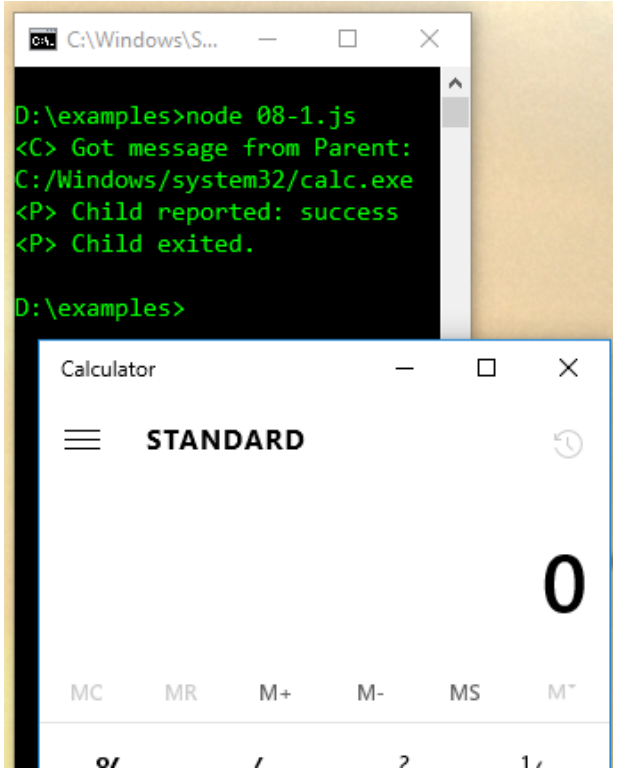
CHILD PROCESS MODULE USAGE EXAMPLE

08-1.js

```
const cp = require('child_process');
const modulePath = './08-2.js';
const child = cp.fork(modulePath);

setTimeout(() => {
    child.send('C:/Windows/system32/calc.exe')
}, 1000);

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

08-2.js

```
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();
    });
});
```



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 master process)

Class Worker:

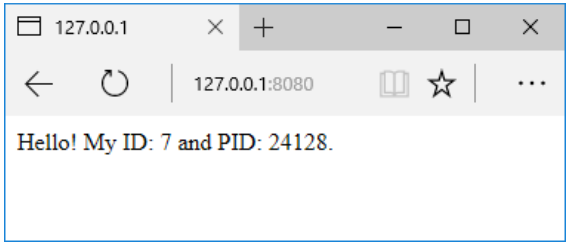
- wraps ChildProcess instance, which was originally created by *child_process.fork()*
- property "id" – unique id for worker, key in *cluster.workers*
- property "process" – ChildProcess instance
- method "send()" - send a message to master (from worker) or to worker (from master)

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(`Hello!
15             My ID: ${cluster.worker.id} and
16             PID: ${process.pid}.`);
17     }).listen(8080);
18
19     console.log(`Worker ${process.pid} started`);
20 }
```

```
D:\examples>node 09.js
Master 14708 is running
Worker 18496 started
Worker 15852 started
Worker 15932 started
Worker 24028 started
Worker 24992 started
Worker 10516 started
Worker 24736 started
Worker 24128 started
```

Cluster automatically recognizes which worker isn't under high load



USEFUL LINKS

- Node.js official FS module documentation
- Node.js official Buffer documentation
- Node.js official Streams documentation
- Node.js official Child Process documentation
- Node.js official Cluster documentation
- Joel Spolsky «The Absolute Minimum Every Software Developer Absolutely, Positively Must Know About Unicode and Character Sets (No Excuses!)»
- Samer Buna «Node.js Streams: Everything you need to know»
- Samer Buna «Node.js Child Processes: Everything you need to know»
- Samer Buna «Scaling Node.js Applications»


NODE.JS GLOBAL
NODE.JS FILESYSTEM AND STREAMS
BY
GENNADII MISHCHENKO