

/YOU DON'T KNOW NODEJS (YET) !



Node.JS ?

- Un environnement d'exécution JavaScript
- Bas niveau, écrit avec un langage de haut niveau

Pourquoi NodeJS ?

2 types d'état de systèmes

- Input/Output (I/O) bound systems
- CPU bound systems
- Plus de I/O bound systems que de CPU

I/O Systems

- Ecrit / Lit dans une base de données
- Appelle un autre un système
- Bref... Différentes tâches qui prennent du temps

Pourquoi NodeJS ?

NodeJS implémente ses interfaces de manière non-bloquante

Comment ?

NodeJS Event Loop (EL)

- Emprunter au moteur V8 de Google
- NodeJS plus ou moins Google Chrome sans interface

Event Loop ?

Emprunter au moteur V8 de Google

Fonctionnalités

- Attend sans cesse des tâches à exécuter puis les délèguent
- Passe un callback en même temps
- Quand la délégation est finie, renvoie une réponse au client

On peut dire que:

NodeJS plus ou moins
Google Chrome sans
interface



```
1 System.out.println("Step 1");  
2 System.out.println("Step 2");  
3 Thread.sleep(1000);  
4 System.out.println("Step 3");
```





```
1 console.log("Step 1");  
2 setTimeout(() => {  
3     console.log("Step 2");  
4 }, 1000);  
5 console.log("Step 3");
```



Pourquoi opter pour les I/O non-bloquants ?

1

Optimisation des ressources disponibles

Principale force de NodeJS

2

Exécution des requêtes

Dans les frameworks multi-thread:
l'exécution d'une requête bloque les suivantes; pas dans NodeJS
(théoriquement)

“Multi-threading is the software equivalent to a nuclear device, if it’s used incorrectly, it can blow up in your face.”

Dans la majorité des cas, les systèmes multi-thread ne sont pas nécessaires

Plus une autre couche de complexité

Par conséquent, les systèmes bloquants doivent être multi-thread.

NodeJS est single thread

...et c'est bien !

Système multi-thread

Avantages

- Performances et concurrences accrues
- Réduit considérablement le nombre d'instance requis
- https://docs.oracle.com/cd/E13203_01/tuxedo/tux71/html/pgthr5.htm

Inconvénients

- Pas facile à écrire
- Pas facile à déboguer
- La gestion de la concurrence n'est pas si facile
- Difficulté à écrire des tests unitaires
- Souvent limitée à une plateforme

Honnêtement, ce n'est pas nécessaire

(sauf dans quelques industries de pointe)

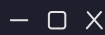
NodeJS est single thread

Élimine toutes les difficultés rencontrées précédemment

Tout n'est pas bloquant en NodeJS, par ex:

**Les instructions
basiques**

**Quelques fonctions
encapsulés dans des
modules**



```
1 console.log("Step 1");
2 for (let index = 0; index < 100000000; index++) {
3   // Take about 100 - 1000 ms, depends on your hardware
4 }
5 console.log("Step 2");
```





```
1  const fs = require('fs');
2  let contents;
3
4  contents = fs.readFileSync('./accounts.txt', 'utf8');
5  console.log(contents);
6  console.log('Hello, Ruby');
7
8  contents = fs.readFileSync('./ips.txt', 'utf8');
9  console.log(contents);
10 console.log('Hello, NodeJS');
11
```





```
1  const fs = require('fs');
2
3  fs.readFile('./accounts.txt', function (error, contents) {
4      console.log(contents)
5  });
6
7  console.log("Hello, Ruby");
8
9  fs.readFile('./ips.txt', function (error, contents) {
10     console.log(contents);
11 });
12
13 console.log("Hello, World");
```



Un grand avantage de NodeJS

JavaScript (TypeScript) everywhere

JavaScript Everywhere

Les types natifs sont (presque) les mêmes

Array, String, Primitives, Functions, Objects

Node's JavaScript != Browser's JavaScript

- Pas la même définition des variables globales: Window vs Global
- Spécificité de NodeJS:
 - `global.module`, `global.require`, `global.process`
 - `process.uptime`, `process.memoryUsage`, `process.cwd`, etc.

**Qui a déjà utilisé / aime /
comprend les callbacks ?**

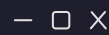
Callback ?

- (Juste) une fonction anonyme qui est passée à l'Event Loop
- Retourne une réponse quand le traitement de la tâche coûteuse est fini

Et le fameux callback hell ?

Fait issu d'une imbrication de
callback successif

<http://callbackhell.com/>



```
1 fs.readdir(source, function (err, files) {
2   if (err) {
3     console.log('Error finding files: ' + err)
4   } else {
5     files.forEach(function (filename, fileIndex) {
6       console.log(filename)
7       gm(source + filename).size(function (err, values) {
8         if (err) {
9           console.log('Error identifying file size: ' + err)
10        } else {
11          console.log(filename + ' : ' + values)
12          aspect = (values.width / values.height)
13          widths.forEach(function (width, widthIndex) {
14            height = Math.round(width / aspect)
15            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
16            this.resize(width, height).write(dest + 'w' + width + '_' + filename, function(err) {
17              if (err) console.log('Error writing file: ' + err)
18            })
19          }.bind(this))
20        }
21      })
22    })
23  }
24 })
```



Les callbacks ...

Permettent de gérer les traitements asynchrones

Mais ... il existe d'autres patterns pour les gérer

- Promise
- Async / Await
- Generators

Encore mais ... ne convient pas à tous les problèmes

Comment faire pour exécuter une tâche plusieurs fois, pas qu'au début ou en fin de traitement, mais au milieu

Event / Event Emitter

Dans la catégorie des **Observer pattern**, inclus dans le core de Node.JS

Pour exploiter cette fonctionnalité, il y a:

- Le Subject
- L'Event Trigger
- L'Event Observer
- Possibilité d'implémenter plusieurs observers pour un seul évènement.

Un article qui m'a bien plus

<https://webapplog.com/node-patterns-from-callbacks-to-observer/>



```
1  const EventEmitter = require('events');
2  const eventEmitter = new EventEmitter();
3
4  // Subject: start
5  // Event Observer
6  eventEmitter.on('start', (date) => console.log('started'));
7
8  // Event trigger
9  eventEmitter.emit('start', new Date());
10
```



On manipule généralement des fichiers dans les serveurs web

Méthode courante: utilisation des buffers
directement

Buffer

**Une classe conçue
pour gérer des
données binaires
brutes**

Limite: environ 1 Go

**Bloque le système
jusqu'à la fin du
traitement**

C'est bien gênant

je vous présente un autre module: Streams

**Objectif: la segmentation
continue des données**

Il y a 4 types de stream:

Readable

Writable

Duplex

A la fois readable
and writable

Transform

qui peuvent à la fois
modifier /
transformer la donnée
en même temps que
celle-ci soit lue ou
écrite


Sans qu'on s'en aperçoive, les streams sont partout

**Requête et
Réponse HTTP**

**Les entrées et
sorties
standards
(stdin et
stdout)**

**Lecture et
écriture de
fichier**

Streams hérite de la classe EventEmitter précédemment abordé (asynchrone)



```
1 process.stdin.resume();
2 process.stdin.setEncoding('utf8');
3
4 process.stdin.on('data', (chunk) => {
5     console.log('chunk:', chunk);
6 });
7
8 process.stdin.on('end', (chunk) => {
9     console.log('--- END ---');
10 });
```

Streams Synchrones



```
1  const readableStream = getReadableStreamSomehow();
2
3  readableStream.on('readable', () => {
4    let chunk;
5    while ((chunk = readableStream.read()) !== null) {
6      console.log('Got %d bytes of data', chunk.length);
7    }
8  });
```

Streams avec les requêtes HTTP

```
1  const http = require('http');
2
3  function transformStream(data) {
4    // write implementation here
5  }
6
7  const server = http.createServer((req, res) => {
8    req.setEncoding('utf8');
9
10     req.on('data', chunk => {
11       transformStream(chunk);
12     });
13
14     req.on('end', (body) => {
15       const data = JSON.parse(body);
16       res.end();
17     });
18   })
19
20  server.listen(3000);
```

**(Express.JS est
implémenté de la même
façon, plus ou moins ...)**

Une dernière façon d'utiliser les streams, les pipes



```
1 let stream1 = fs.createReadStream('file.txt');  
2 let stream2 = zlib.createGzip();  
3 let stream3 = fs.createWriteStream('file.txt.gz');  
4  
5 stream1.pipe(stream2).pipe(stream3);
```

Les pipes Node.JS :

Permettent de chaîner des processus

la sortie d'un processus sert d'entrée à un autre

Implémenter de la même façon en NodeJS

la sortie d'un stream sert d'entrée à une autre

Le type utilisé par les streams pour manipuler les données est ... le buffer

Les différentes façons de créer un buffer:

- `Buffer.alloc(size)`
- `Buffer.from(array)`
- `Buffer.from(buffer)`
- `Buffer.from(string, [, encoding])`

En réalité, tout est transmis sous forme de buffer (requête HTTP, manipulation du système de fichier, etc.)

La conversion en stream se fait en spécifiant un encodage

Un livre pour approfondir les streams

(<https://github.com/substack/stream-handbook>)

**Si vous maitrisez les streams, vous serez des
guru de NodeJS**

**Comment mettre en
échelle / scaler un
système single-thread ?**

Remember it's single-threaded ?

Certains développeurs de système multi-thread (comme Java) sous-entendent que NodeJS n'exploite pas toutes les ressources disponibles, c'est **FAUX**.

La performance est souvent un faux problème

Les opérations asynchrones sont exécutées sur des différents threads, credit goes to Event Loop architecture.

Ils sont délégués à l'Event Loop

Comment ?

Le module **cluster**

Un processus master

- démarre un worker / processus fils
- écoute si ces workers sont actuellement en train de fonctionner, morts, etc.
- redémarre un worker

Toutefois, ce n'est pas le module idéal

Il est plus ou moins maigre et moyen

On utilise le plus souvent des librairies

strong-cluster-control

Pm2 (mon choix perso et celui de beaucoup d'autres développeurs)

- est un load-balancer (technologie conçue pour distribuer la charge de travail entre différents serveurs / applications)
- 0 s de temps de rechargement / Forever alive
- a un bon rapport de test coverage
- pas de nécessité de modifier le code source
- encore d'autres fonctionnalités

Toutefois, si vous décidez d'utiliser la librairie standard

fork

- démarre des nouveaux processus NodeJS
- nouvelle instance V8 créée
- workers multiples

spawn

- démarre des nouveaux processus (catégorie confondues)
- utiliser le plus souvent pour de large volume de données comme les streams
- pas de nouvelle instance de V8 créée

Toutefois, si vous décidez d'utiliser la librairie standard

exec

- utiliser le plus souvent pour les buffers, `async / await`
- utiliser pour récupérer la donnée entière en une seule fois
- donc des petites commandes

Prochaine partie: la gestion des erreurs asynchrones

Pourquoi gérer les erreurs ?

la combinaison de multiples facteurs mènent toujours à de multiples bugs

- périmètre de l'application
- complexité de l'application
- facteurs externes à l'application

Pourquoi gérer les erreurs ?

Donc

- Permet à l'utilisateur que quelque chose s'est mal passée ... et de manière conviviale
- Ajouter quelques subtilités à déboguer les autres ou vous-mêmes votre programme

D'abord, la gestion d'erreur des I/O bloquants



```
1 function aFunctionThatAlwaysFail() {  
2     throw new Error('Fail');  
3 }  
4  
5 try {  
6     aFunctionThatAlwaysFail();  
7 } catch (error) {  
8     console.log(`Custom error: ${error}`);  
9 }
```

La gestion d'erreur des I/O non-bloquants



```
1  try {  
2      setTimeout(() => {  
3          throw new Error();  
4      }, 0);  
5  } catch(e) {  
6      console.log(e); // Nothing  
7  }
```

try catch est inutile pour les
codes non-bloquants

**ils sont délégués à l'Event Loop (exécuté
hors du contexte du try catch)**

Comment faire ?

Aucune solution efficace (callback)

Ecouter l'évènement **uncaughtException**

- Doit être utilisé avec beaucoup de précaution ou
- En dernier recours



```
1 process.on('uncaughtException', function(err) {  
2     console.error('uncaughtException', err.message);  
3     console.error(err.stack);  
4     process.exit(1);  
5 });  
6
```



C++ addons

Bonus

Beaucoup de personnes pensent que NodeJS est
uniquement du JavaScript

NodeJS en soi est écrit avec plus de C++ et
de C.

Comment ?

Objectifs

**Effectuer des calculs
intensifs, parallèles et
de haute précision**

**Utiliser des
bibliothèques C++
dans NodeJS**

Créer le fichier C++ (hello.cc)

```
1 #include <node.h>
2
3 namespace demo {
4
5 using v8::FunctionCallbackInfo;
6 using v8::Isolate;
7 using v8::Local;
8 using v8::Object;
9 using v8::String;
10 using v8::Value;
11
12 void Method(const FunctionCallbackInfo<Value>& args) {
13     Isolate* isolate = args.GetIsolate();
14     args.GetReturnValue().Set(String::NewFromUtf8(
15         isolate, "world").ToLocalChecked());
16 }
17
18 void Initialize(Local<Object> exports) {
19     NODE_SET_METHOD(exports, "hello", Method);
20 }
21
22 NODE_MODULE(NODE_GYP_MODULE_NAME, Initialize)
23
24 }
```

Créer le fichier binding.gyp



```
1 {  
2     "targets": [  
3         {  
4             "target_name": "addon",  
5             "sources": ["hello.cc"]  
6         }  
7     ]  
8 }
```

Et ...

```
npm install -g node-gyp  
node-gyp configure  
node-gyp build
```

(Produit un dossier build/Release/addon)

Dans un nouveau fichier ...



```
1  const addon = require('./build/Release/addon');  
2  console.log(addon.hello());
```

“Any application that can be written in JavaScript will eventually be written in JavaScript.”

Atwood's law, écrit en 2007 dans son blog