NODEJS 1

[**Mono thread**](#_bs1pfgpc1d0e) **3**

[**Asynchrone et async.js**](#_h44qgiieaakk) **9**

[Limite de l’asynchrone](#_kgo5v27z3kja) 9

[Async Waterfall](#_9717b3fm3jop) 14

[Async Parallel](#_4d6ihk2xm0n) 17

[Async Loop](#_6wbnf8ckjjz3) 19

[Conclusion async](#_8ve45aii7ig0) 21

[**Conclusion**](#_hkoe2t3ljzyk) **22**

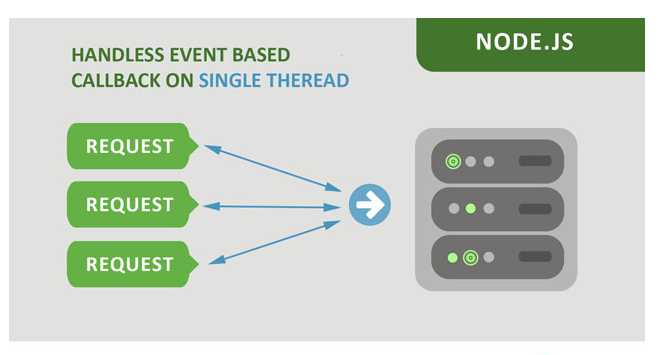
# Mono thread

Node à la particularité d’être un langage mono threadé, à la différence de PHP, un seul processus mono thread tourne pour récupérer et traiter l’ensemble des requêtes. A l’inverse dans un serveur classique Apache/PHP le processus php va se diviser en plusieurs thread permettant ainsi de traiter plusieurs requêtes (1 par thread), les autres étant mise en attente.

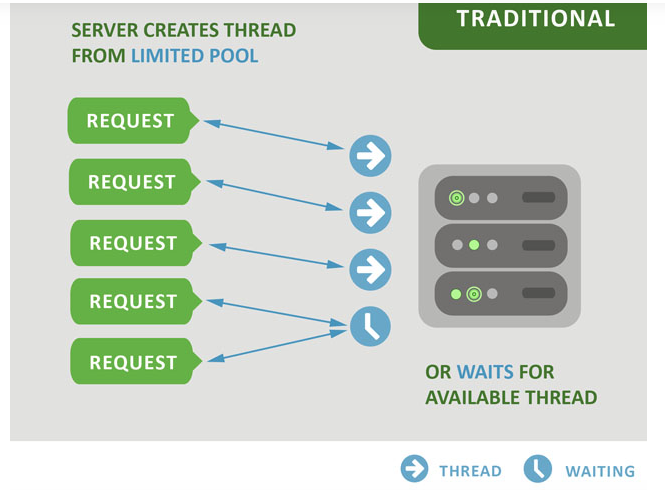
De ce fait, node traite toutes les requêtes en même temps, il change de requête dès qu’une entrée sortie survient (requête HTTP, lecture d’un fichier, interaction base de donnée).

Alors que PHP divise simplement les requêtes sur nombre de thread et met les autres en attentes.

**NodeJS :**



**PHP:**



Plus concretement on peut visualiser ce qu’il se passe de cette façon. Imaginons un serveur PHP et un serveur node contenant chacun la route GET /db qui permet de récupérer le contenu d’une table en base de données de faire la somme des éléments et de renvoyer le résultat. Imaginons que 3 requêtes sont envoyées en même temps au serveur et que PHP fonctionne sur 2 theads.

PHP traitera les requêtes de cette façon (les blocs verts sont réalisés en parallèles):

* Envoi requête 1 sur thread 1  
  1. Envoi requête bdd
  2. Attente réponse bdd
  3. Récupération réponse bdd
  4. Somme
  5. Renvoi du résultat
  6. Fin process, passage à la requête suivante
* Envoi requête 2 sur thread 2  
  1. Envoi requête bdd
  2. Attente réponse bdd
  3. Récupération réponse bdd
  4. Somme
  5. Renvoi du résultat
  6. Fin process, passage à la requête suivante
* Mise en attente de requête 3 en attente

A l’inverse node traitera ces requête de la façon suivante (pour plus lisibilité la requête 1 est en rouge, la 2 en vert et la 3 en bleue, les entrées sorties = changement de requête sont en jaune):

1. Récupération requête 1
2. Mise en attente requête 2 dans event loop
3. Mise en attente requête 3 dans event loop
4. Requête 1 : Envoi requête BDD
5. Entrée / Sortie => Changement de requête (prochaine action dans event loop)
6. Requête 2 : Envoi requête BDD
7. Entrée / Sortie => Changement de requête (prochaine action dans event loop)
8. Requête 3 : Envoi requête BDD
9. Entrée / Sortie => Changement de requête (prochaine action dans event loop)
10. Requête 2 : réponse bdd
11. Requête 2 : sommez
12. Requête 2 : renvoi résultat
13. Entrée / Sortie => Changement de requête (prochaine action dans event loop)
14. Requête 1 : réponse bdd
15. Requête 1 : somme
16. Requête 1 : renvoi résultat
17. Entrée / Sortie => Changement de requête (prochaine action dans event loop)
18. Requête 2 : fin requête
19. Requête 3 : réponse bdd
20. Requête 3 : somme
21. Requête 3 : renvoi résultat
22. Entrée / Sortie => Changement de requête (prochaine action dans event loop)
23. Requête 1 : fin requête
24. Requête 3 : fin requête

La différence est donc la suivante, lors du temps d’attente lors d’une entrée / sortie (attente réponse bdd, attente réponse http, attente lecture fichier, ….), node est capable de d’effectuer d’autres actions.

Un exemple plus parlant pourrait être la réalisation de 2 requête HTTPs.  
  
**En PHP:**

$ch = curl\_init();

curl\_setopt($ch, CURLOPT\_URL, “http://www.google.fr/”);

curl\_setopt($ch, CURLOPT\_HEADER, TRUE);

curl\_setopt($ch, CURLOPT\_NOBODY, TRUE); // remove body

curl\_setopt($ch, CURLOPT\_RETURNTRANSFER, TRUE);

$head = curl\_exec($ch);

$httpCode = curl\_getinfo($ch, CURLINFO\_HTTP\_CODE);

curl\_close($ch);

echo “requete 1 ok”;

$ch = curl\_init();

curl\_setopt($ch, CURLOPT\_URL, “http://www.amazon.fr/”);

curl\_setopt($ch, CURLOPT\_HEADER, TRUE);

curl\_setopt($ch, CURLOPT\_NOBODY, TRUE); // remove body

curl\_setopt($ch, CURLOPT\_RETURNTRANSFER, TRUE);

$head = curl\_exec($ch);

$httpCode = curl\_getinfo($ch, CURLINFO\_HTTP\_CODE);

curl\_close($ch);

echo “requete 2 ok”;

**En node :**

request.get({url: “<http://www.google.fr/>”}, function(err, response, body){

console.log(“requete 1 ok”);

});

request.get({url: “[http://www.amazon.fr/](http://www.google.fr/)”}, function(err, response, body){

console.log(“requete 2 ok”);

});

En php le code va être exécuté séquentiellement sur le thread, “requête 2 ok” sera toujours affiché après “requête 1 ok”.

En node, lors de l’envoi de la première requête (entrée / sortie), le thread ne va pas se bloquer en attendant la réponse et va exécuter les instructions suivante, ainsi, si la seconde requête met moins de temps à s'exécuter il est possible que “requête 2 ok” soit affichée avant “requête 1 ok”. Il est ainsi possible de paralléliser les entrées / sorties d’un processus afin d’utiliser intelligemment les temps d’attentes liés aux entrées / sorties.

Il n’est cependant pas toujours possible de paralléliser les opérations, par exemple si l’envoi de la seconde requête dépend du résultat de la première. En PHP le code reste inchangé cependant en node nous allons devoir déplacer la requête 2 dans le callback de la requête 1 afin qu’elle s’execute uniquement après récupération du résultat :

request.get({url: “<http://www.google.fr/>”}, function(err, response, body){

console.log(“requete 1 ok”);

request.get({url: “[http://www.amazon.fr/](http://www.google.fr/)”}, function(err, response, body){

console.log(“requete 2 ok”);

});  
});

Ce code fonctionnera de la même manière que le code PHP, “requête 2 ok” sera toujours affiché après “requête 1 ok”, cependant pendant les entrée / sorties liées aux requêtes HTTP, node sera capable de traiter d’autres requêtes en même temps.

L’asynchrone est la particularité de node et doit être géré intelligemment afin d’éviter de bloquer le thread principale au risque de ralentir l’ensemble du serveur. Il est bien sur possible de forcer node à utiliser des version synchrone (séquentielles) cependant cela aura pour effet de ralentir tout le code.

Exemple, lecture de 2 fichiers, en node standard asynchrone:

fs.readFile(PATH, function(err, data){

console.log(“File 1 ok”);

fs.readFile(PATH2, function(err, data){

console.log(“File 2 ok”);

});

});

Cette version peut être transformée en version synchrone, cependant ce code aura pour effet de bloquer le thread durant les entrées / sorties, une requête arrivant sur le serveur sera donc mise en attente le temps de finir la lecture des 2 fichiers :

var data1 = fs.readFileSync(PATH);

console.log(“File 1 ok”);

var data2 = fs.readFileSync(PATH2);

console.log(“File 2 ok”);

Le synchrone peut donc être utiliser, pour des programmes locales notamment, ou dans des cas extrêmement rare (initialisation du serveur), mais **jamais** dans le cadre du traitement d’une requête au risque de baisser drastiquement (x100) les performances du serveurs. L’asynchrone est la particularité de node, et l’une de ces principales forces, il doit être bien manipulé pour être efficace.

# Asynchrone et async.js

## Limite de l’asynchrone

L’asynchrone peut poser des problèmes d’écritures dans le cadre d’exécution de processus complexe. De plus il peut être très difficile à lire dès lors qu’il y a plusieurs callback d’imbriqués.

Prenons l’exemple ci-dessous. Nous voulons effectuer une requête en base de donnée, puis une requête web à partir du résultat obtenu en base de donnée, puis écrire un fichier contenant le résultat de la requête web. Enfin si une erreur survient nous souhaitons arrêter le process, sinon en fin de processus nous voulons afficher fin.

Le code node est donc le suivant :

EXEMPLE 1

var name = “nom”;

MaCollection.findOne({\_name: name}, function(err, object){

if(err || !object)

{

console.log(“Erreur”);

exit(0);

}

else

{

request.get({url: object.url}, function(err, response, body){

if(err || response.statusCode != 200 || !body)

{

console.log(“Erreur”);

exit(0);

}

else

{

fs.writeFile(PATH, body, function(err){

if(err)

{

console.log(“Erreur”);

exit(0);

}

else

{

console.log(“FIN”);

}

});

}

});

}

});

Même si ce code reste lisible, on peut imaginer la difficultée de lecture lorsque l’on se retrouve avec 10 callback imbriqués les uns dans les autres.

Egalement on peut identifier 2 autres problèmes, le premier concerne la gestion de la parallélisation.

Par exemple, nous souhaitons réaliser 2 requêtes HTTP, une sur google.fr, l’autre sur amazon.fr, puis afficher “fin” lorsque les deux requêtes sont terminées. L’exécution de la seconde requête ne dépendant pas du résultat de la première nous pouvons donc les traiter en parallèle :

request.get({url: “<http://www.google.fr/>”}, function(err, response, body){

console.log(“requete 1 ok”);

});

request.get({url: “[http://www.amazon.fr/](http://www.google.fr/)”}, function(err, response, body){

console.log(“requete 2 ok”);

});

Cependant ne sachant pas laquelle va terminer avant l’autre, il est difficile de savoir ou placer notre console.log(“fin”). La seule façon de s’assurer qu’il soit executé uniquement à la fin des deux requêtes est la suivante :

EXEMPLE 2

var done = 0;

request.get({url: “<http://www.google.fr/>”}, function(err, response, body){

console.log(“requete 1 ok”);

// quand on a traiter les deux requêtes

if(++done == 2)

{

console.log(“fin”);

}

});

request.get({url: “[http://www.amazon.fr/](http://www.google.fr/)”}, function(err, response, body){

console.log(“requete 2 ok”);

// quand on a traiter les deux requêtes

if(++done == 2)

{

console.log(“fin”);

}

});

On peut voir que l’on est obligé de dupliquer le bloc final dans chacun des appels parallèles, ce qui peut également devenir très compliqué lorsqu’il s’agit d’en faire 50 et que le bloc final comporte des centaines d’instructions.

Enfin le dernier problèmes concerne l’exécution séquentielle de requête asynchrone. Nous allons prendre l’exemple suivant : Imaginons un tableau javascript contenant X url (nous ne connaissons) pas le X, pour lequel nous voulons pour chaque url récupérer la page html associées via une requête HTTP. La seule façon de réaliser cela sera en récursion asynchrone :

EXEMPLE 3

var table = [“<http://www.google.fr>”, “<http://www.amazon.fr>”, “<http://lemonde.fr/>”, …];

Var result = {};

function execute(table, err, finalCallback)

{

if(!err && table.length > 0)

{

request.get({url: table[0]}, function(err, response, body){

if(err)

{

return finalCallback(err);

}

else

{

result[table[0]] = body;

// on enlève l’url que l’on vient de traiter

table.splice(0, 1);

return execute(table, null, finalCallback);

}

});

}

else

{

return finalCallback(err);

}

}

execute(table, null, function(err){

if(err)

console.log(“Error”);

else

console.log(“Fin”);

});

Il est maintenant facile d’imaginer la difficulté d’imbriquer des loops dans des waterfalls dans des parallèles.

La gestion de l’asynchrone ne se fait donc pas à la main, même si cela est possible, car cela peut s’avérer très compliqué pour des processus complexe et la gestion d’erreurs peut très rapidement s’avérer impossible.

Nous utilisons donc un module dédié à la gestion de l’asynchrone, ce module se nomme “async”. Nous allons voir comment transformer les processus précédemment détaillé avec ce plugin.

3 fonctions de async vont particulièrement nous intérésser :

**async.waterfall => exemple 1**

**async.parallel => exemple 2**

**async.eachOfLimit => exemple 3**

## Async Waterfall

async.waterfall permet de gérer facilement l’imbrication de callbacks, son prototype est le suivant :

async.waterfall([

initialisation,

action1,

action2,

action3,

...

], callbackfinal)

Le module va exécuter séquentiellement initialisation => action 1 => action 2 => action 3, une fois toutes les actions finis, le module va appeler la fonction callback final. Si une erreur survient (par exemple dans action2), le callback final est appelé avec une erreur (et action3 ne sera pas exécutée).

*Le prototype de la fonction d’initialisation est le suivant :*

**function(wcb)** // wcb est le callback permettant de passer à l’action suivante, il prend en paramètre (err, datas), si err est définis le processus est interrompu et on passe dans callback final, sinon l’action suivante est appelé avec datas en paramètre.

*Le prototype des fonctions actions est le suivant :*

**function(wdatas, wcb)** // wdatas est les donnée renvoyées par l’action précédente, wcb est le callback permettant de passer à l’action suivante

Le prototype du callback final est le suivant :

**function(err, result)** // err contient une erreur si le processus a été interrompu avant la fin, sinon les données renvoyées par la dernière action sont présentes dans result

Notre EXEMPLE 1, se transforme donc de la façon suivante :

async.waterfall([

// fonction d’initialisation

function(wcb)

{

var wdatas = {

name: “nom”

};

return wcb(null, wdatas);

},

read\_collection, // action 1

get\_request, // action 2

write\_file // action 3

], function(err, result){ // callback final

if(err)

{

console.log(“[error][waterfall]”+err);

exit(0);

}

else

{

console.log(“Fin”);

exit(0);

}

});

function read\_collection(wdatas, wcb) // action 1

{

var name = wdatas.name;

MaCollection.findOne({\_name: name}, function(err, object){

if(err || !object)

{

// on interrompt le processus, appel du callback final avec une erreur

console.log(err);

return wcb(“[read\_collection]”+err, wdatas);

}

else

{

// on enregistre l’objet dans les données, puis on passe à l’action suivante en envoyant les données

wdatas.object = object;

return wcb(null, wdatas);

}

});

}

function get\_request(wdatas, wcb) // action 2

{

var object = wdatas.object;

request.get({url: object.url}, function(err, response, body){

if(err || response.statusCode != 200 || !body)

{

// on interrompt le processus, appel du callback final avec une erreur

console.log(err);

return wcb(“[get\_request]”+err, wdatas);

}

else

{

// on enregistre l’objet dans les données, puis on passe à l’action suivante en envoyant les données

wdatas.body = body;

return wcb(null, wdatas);

}

});

}

function write\_file(wdatas, wcb) // action 3

{

var body = wdatas.body;

fs.writeFile(“./file1”, body, function(err){

if(err)

{

// on interrompt le processus, appel du callback final avec une erreur

console.log(err);

return wcb(“[write\_file]”+err, wdatas);

}

else

{

return wcb(null, wdatas);

}

});

}

On peut voir que bien que plus long, ce code est facilement maintenable, est divisé en fonction simple, et permet une bonne gestion des erreurs. De plus il est facilement modifiable et il serait donc particulièrement facile de rajouter une action entre les actions 2 et 3.

C’est pourquoi, dès lors qu’une fonction est amenée à évoluer, ou qu’elle contient au moins deux callback imbriquées on utilisera ce templating.

## Async Parallel

De la même façon, async contient un template permettant de réaliser X actions en parallèle, une fois ces actions exécutées un callback final est appelé avec les résultats. Son prototype est le suivant :

async.parallel({

nomaction1: action1

nomaction2: action2

...

}, callbackFinal);

Le prototype des actions est le suivants :

**function(pcb)** // ou pcb est une fonction prenant en paramètre (err, data)

Le prototype du callbackFinal est le suivant :

**function(err, result)** // ou err contient une erreur si le processus à été interrompu et result contient les résultats renvoyés par les actions (le résultat de chaque action est présent à la clef “nomactionX”, par exemple, le résultat de action1 sera présent dans result.nomaction1

Nous pouvons donc transformer l’exemple 2 de la façon suivante :

async.parallel({

amazon: get\_amazon

google: get\_google

}, function(err, result){

if(err || !result)

{

console.log(“[erreur][parallel]”+err);

exit(0);

}

else

{

// result.amazon contient le body récupéré dans get\_amazon

// result.google contient le body récupéré dans get\_google

console.log(“fin”);

}

});

function get\_amazon(pcb)

{

request.get({url: “http://www.amazon.fr/”}, function(err, response, body){

if(err || response.statusCode != 200 || !body)

{

console.log(err);

// interruption du flow, appel de callback final avec une erreur

return pcb(“[get\_amazon]”+err, null);

}

else

{

// renvoi du résultat

return pcb(null, body);

}

});

}

function get\_google(pcb)

{

request.get({url: “http://www.google.fr/”}, function(err, response, body){

if(err || response.statusCode != 200 || !body)

{

console.log(err);

// interruption du flow, appel de callback final avec une erreur

return pcb(“[get\_amazon]”+err, null);

}

else

{

// renvoi du résultat

return pcb(null, body);

}

});

}

Lorsque plusieurs actions pourront être réalisées en même temps (elle ne dépende pas les unes des autres), par exemple récupération dans X tables différéntes de Y objets en base de données, on utilisera systématiquement ce template.

## 

## Async Loop

Enfin il est possible d’appliquer facilement des fonctions asynchrones sur chacun des élements d’un tableau avec la fonction async.eachOfLimit, cette fonction à la particularité de permettre de faires ces opérations sur un élément à la fois, ou alors sur 2, 4, 5, … à la fois.

Son prototype est le suivant

async.eachOfLimit(table, number, action, finalCallback);

Ses arguments sont les suivants :

* **table** doit être un tableau javascript vide ou contenant des éléments.
* **number** est le nombre d’opérations que l’on veut faire en même temps, 1 pour exécuter les opérations séquentiellement les unes à la suite des autres, 2 pour traiter 2 éléments à la fois en parallèle, etc… La seule limite étant la puissance de l’ordinateur, si l’on doit récupérer le contenu de 1000 URLs on peut imaginer effectuer les requêtes par batch de 10, cependant si l’on doit faire des grosses opérations en base de données il vaudra mieux les exécuter une par une afin d’éviter de saturer la ram et le CPUs.
* **action** est la fonction asynchrone à executer sur chaque élément, son prototype est le suivant:  
    
  **function(element, index, ecb) //** element est l’élement courant du tableau que l’on traite, index est l’index de l’élement dans le tableau, et ecb est le callback à appelé une fois que l’on a traité l’élément il renvoi ou non une erreur
* **finalCallback** est le callback final appelé une fois tout le tableau traité, son prototype est le suivant :  
    
  **function(err)**

On peut donc facilement transformer notre EXEMPLE 3 de la façon suivante :

var table = [“<https://www.google.fr>”, “[https://amazon. fr](https://amazon.fr)”, “<https://perdu.com>”, …];

var result = {};

// 1 élement à la fois, possible de mettre 2, 3, 4 , ….

async.eachOfLimit(table, 1, function(element, index, ecb){

get\_request(element, function(err, body){  
 result[element] = body;

return ecb(err);

});

}, function(err){

if(err)

{

console.log(“[erreur][eachoflimit]”+err);

exit(0);

}

else

{

// result[“<https://www.google.fr>”] contient la page html de google

// result[“<http://amazon.fr>”] contient la page html de amazon, ...

console.log(“fin”);

}

});

function get\_request(url, cb)

{

request.get({url: url}, function(err, response, body){

if(err || response.statusCode != 200)

{

console.log(err);

return cb(“[get\_request][“+element+”]”+err, null);

}

else

{

return cb(null, body);

}

});

}

## Conclusion async

Il est essentiel d’apprendre à utiliser ces 3 fonctions correctement, ce sont elles qui permettent de structurer le code.

Il est très facile de faire quelque chose qui marche en nodejs, cependant le faire proprement et de manière optimisée est impératif pour que la correction de bug et l’évolution du code ne devienne pas impossible

# Conclusion

Le node js est un langage qui repose sur l’asynchrone, sa gestion peut être la force ou la faiblesse de l’application. Nous avons vu que certains modules (ici async), permet de simplifier la gestion de ces appels asynchrone.

De nombreux modules permettent de gérer tout un tas de chose en nodejs (lecture fichier, requête web, encryption/décryption, etc…), il est important de sélectionner ceux utilisés avec soin (nombre de personnes à maintenir le module, version du modules, etc…) car on peut en trouver 1 millions qui font la même choses.

Une bonne application dépendra avant tout des modules qu’elle utilise.