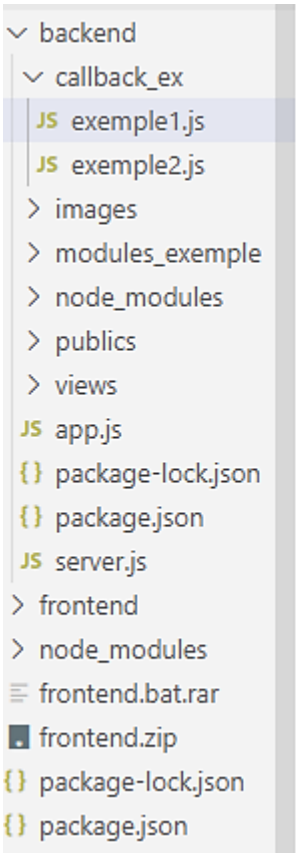
CALLBACK

Concept : vous venez dans un magasin pour acheter quelque chose que vous aimez. Le personnel de la boutique vous dit que l'article est épuisé maintenant. Vous laissez le numéro de téléphone et demandez- lui de vous rappeler immédiatement après que la marchandise est disponible. Ensuite, vous pouvez sortir ou faire un travail et n'avez plus besoin de vous soucier de la boutique, jusqu'à ce que vous receviez un appel téléphonique pour signaler l'article demandé par la boutique.

Le serveur NodeJS peut recevoir plusieurs demandes (request) de différents utilisateurs. Donc afin d'améliorer le fonctionnement, tous les API du NodeJS sont conçu pour soutenir Callback. Le "callback" est une fonction (function), elle sera appelée lorsque le NodeJs achève une tâche (task) précise.

Exemple : Blocking




```
5   // callback_ex/exemple1.js
6
7   var fs = require("fs");
8
9   // -----> Lecture fichier1:
10  console.log("\n");
11  console.log("Lecture du fichier1.txt");
12
13  var data1 = fs.readFileSync('C:\\test\\fichier1.txt');
14  console.log("- Data du fichier1 : ");
15  console.log(data1.toString());
16
17  // -----> Lecture fichier2:
18  console.log("\n");
19  console.log("Lecture fichier2.txt");
20
21  var data2 = fs.readFileSync('C:\\test\\fichier2.txt');
22  console.log("- Data du fichier 2: ");
23  console.log(data2.toString());
24
25  console.log("\n");
26  console.log("Programme terminé");
```

```
C:\ASAKO\1DEV\1 TECHNO WEB\COURS NODEJS\tp\fullstack\backend>node ./callback_ex/exemple1.js


Lecture du fichier1.txt
- Data du fichier1 :
bonjour fichier1



Lecture fichier2.txt
- Data du fichier 2:
bonjour fichier2



Programme terminé
```

Dans le NodeJS, des API sont conçu pour soutenir le Callback. Ce programme précédent lit 2 fichiers en utilisant le module **fs**, qui donne deux fonctions telles que ***readFile*** et ***readFileSync*** pour lire le fichier.

Le ***readFileSync*** est une fonction qui lit les fichiers de manière synchrone (synchronous). Par conséquent, pendant que cette fonction est en cours d'exécution, cela empêchera (block) le programme d'exécuter les lignes de code suivantes.

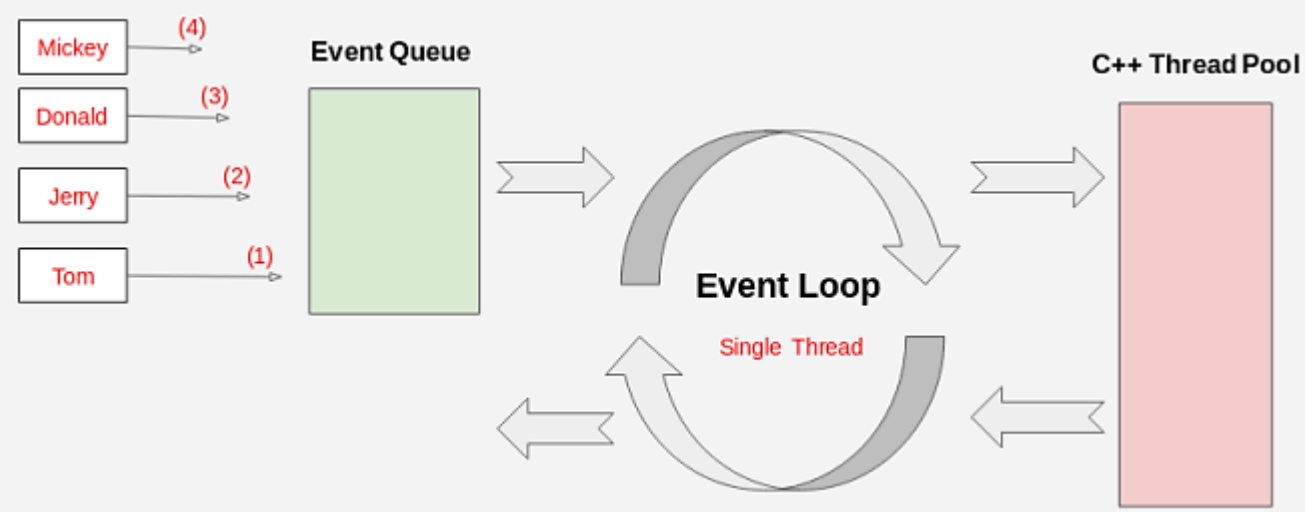
Exemple : Non Blocking

```
// callback_ex/exemple2.js
var fs = require("fs");
// fonction Callback
function readFinishedFile1(err, data)  {
    if (err) console.log(err);
    console.log("- Data du fichier1: ");
    console.log(data.toString());
}

// fonction Callback
function readFinishedFile2(err, data)  {
    if (err) console.log(err);
    console.log("- Data du fichier2: ");
    console.log(data.toString());
}

// -----> Lecture fichier1:
console.log("\n");
console.log("Lecture fichier1.txt");
fs.readFile('C:\\test\\fichier1.txt', readFinishedFile1);

// -----> Lecture fichier2:
console.log("\n");
console.log("Lecture du fichier2.txt");
fs.readFile('C:\\test\\fichier2.txt', readFinishedFile2);

console.log("\n");
console.log("Programme terminé");
```

```
C:\ASAKO\1DEV\1 TECHNO WEB\COURS NODEJS\tp\fullstack\backend>node ./callback_ex/exemple2.js

Lecture fichier1.txt


Lecture du fichier2.txt


Programme terminé
- Data du fichier2:
bonjour fichier2

- Data du fichier1:
bonjour fichier1
```

NODEJS EVENT LOOP

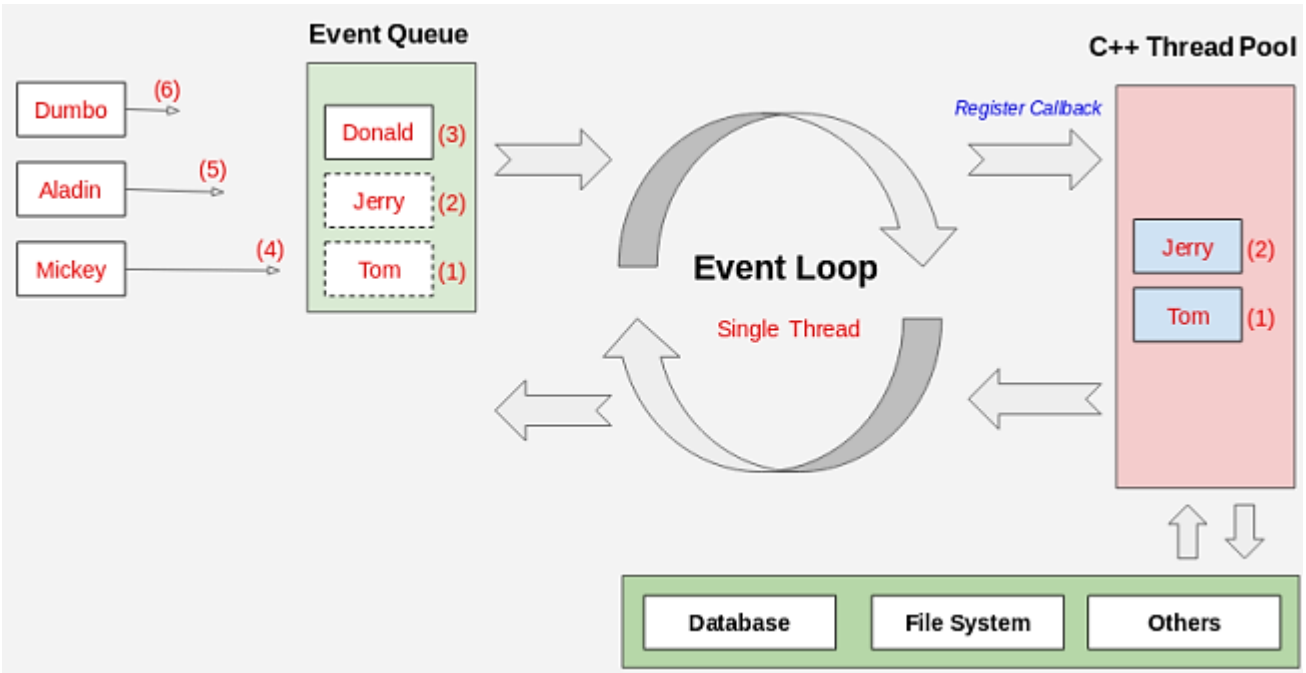
Event Queue : NodeJS est une application de ***thread unique*** (***Single Thread***), qui fonctionne sur une plateforme écrite en C++. Cette plateforme utilise le multi-thread pour effectuer des tâches en même temps.



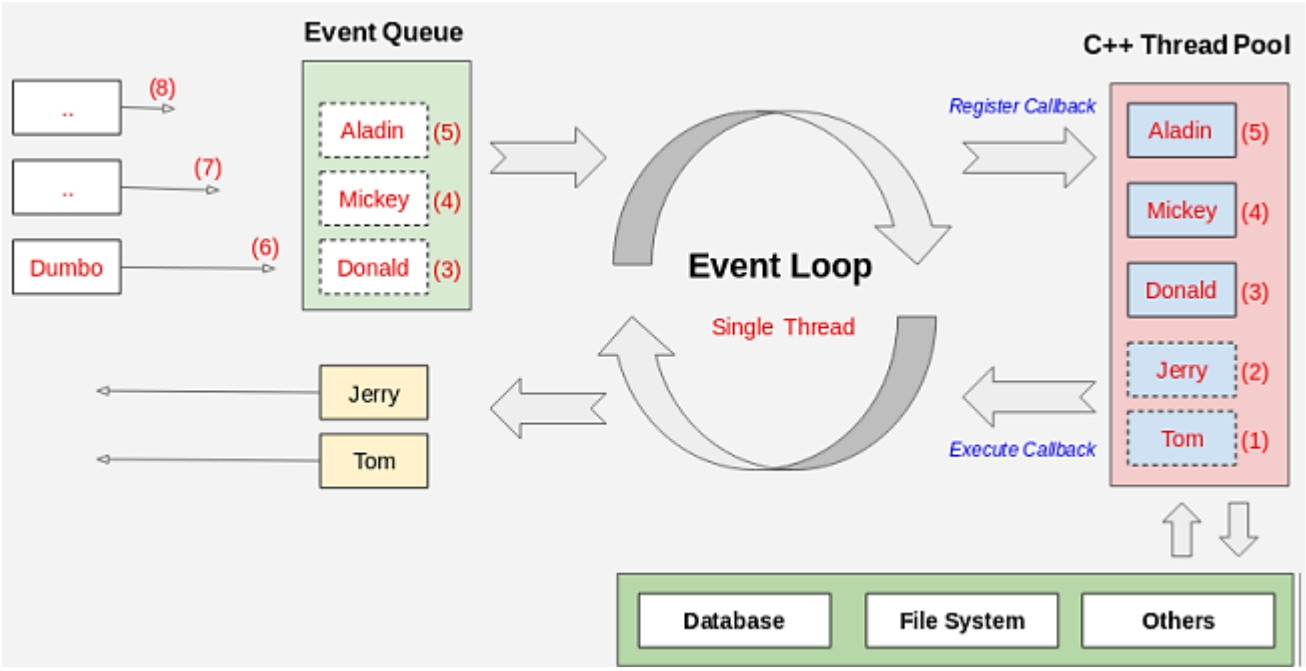
Chaque demande (request) de l'utilisateur est traitée comme un événement (event) par le NodeJS. Elles sont placées dans un Event Queue (La liste des événements). Le NodeJS utilise le principe FIFO (First In First Out).

Event Loop : Comme une boucle infinie, elle transmet les demandes au Thread Pool (Pool de threads) et chaque requête est enregistrée une fonction Callback. Quand une requête est terminée, la fonction Callback correspondante sera appelée à être exécuté.

Thread Pool : Étant un programme écrit par le langage C++, il soutient les multi-threads (Multi Threads). Donc, des demandes seront traitées sur différent threads. Le NodeJS fournit également des processus multiples (Multi Processes). Cela signifie qu'ils peuvent être exécutés sur des coeurs (Core) différents.



Lorsque le traitement d'une demande est terminé, le NodeJS appelera la fonction Callback (Enregistré pour cette demande) pour l'exécuter.



Conclusions :

- Si chaque connexion du Server ouvert un thread, il prendra beaucoup de mémoire. Ceci a été prouvé lorsqu'on compare Apache et Nginx (Les deux Web Server déploient des applications PHP). Le Apache utilise plus de mémoire que le Nginx. En fait, le NodeJS est similaire au Nginx car ils utilisent qu'un seul thread (Single thread) pour recevoir des connexions de l'utilisateur et traiter chaque demande de l'utilisateur comme un événement.

- Des activités I/O coûtent beaucoup de ressources du système, donc le NodeJS gère l'utilisation des activités I/O strictement. Par conséquent, on doit utiliser une Callback quand on exécute les devoirs relatifs à I/O.

À la base, plusieurs choses dans le NodeJS exécutent parallèlement sur différent threads mais elles sont gérées directement par le NodeJS.

NODEJS EVENTEMITTER

Exemple évènement : Ouvrir un fichier

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

// Creation d'un flux du fichier à lire
var rs = fs.createReadStream('C:/test/exemple.txt');

// Evenement open
rs.on('open', function() {
    console.log('Fichier ouvert!');
});
```

Le module **events** fournit une classe EventEmitter, qui est une classe centrale dans le NodeJS aidant à définir un événement, inscription Listener (Auditeur) de cet événement et émettant (emit) l'événement.
Le **Listener** est une fonction. Elle sera appelée pour exécuter lorsqu'un événement se déroule. Probablement, il y a 0, 1 ou plusieurs Listener are attachés (bind) à cet événement.

Les classes EventEmitter

Méthode	Description
addListener(event, listener)	Ajoutez un **listener** à la fin du tableau de **listeners** pour l'événement précis. Cette méthode ne vérifie si ce **listener** a été ajouté ou non.
on(event, listener)	Cette méthode est identique exactement 100% à celle de **addListener**.
once(event, listener)	Ajoutez un **listener** au tableau **listeners** de l'événement précis. Mais ce **listener** a été appelé une seul fois lorsqu'un événement se déroule.Puis il est enlevé du tableau.
removeListener(event, listener)	Enlevez un **listener** du tableau des **listeners** de l'événement précis. Si un **listener** a été ajouté à ce tableau plusieurs fois, afin d'enlever ce **listener** vous devez appeler cette méthode plusieurs fois.
removeAllListeners([event])	Enlevez tous les **listener**, ou supprimez tous les **listener** de l'événement précis.
setMaxListeners(n)	Par défaut, **EventEmitter** imprimera un avertissement si plus de 10 **listener** ont été ajoutés à un événement précis. C'est un défaut util qui permet de trouver des fuites de mémoire (memory leaks). Bien évidemment, vous pouvez personnaliser un autre numéro ou le mettez 0 si vous voulez qu'il est illimité (unlimited).
listeners(event)	Renvoie un tableau de **listener** à l'événement précis.
emit(event, [arg1], [arg2], [...])	Exécutez chaque **listener** dans le tableau, avec des paramètres. Renvoie **true** si le tableau a au moins un **listener**, si non, renvoie false.

Exercice : Créez un fichier exercice.js. Dans ce fichier :

- Importez le module events et créer un objet EventEmitter.
- Définir 3 listeners :

✓ Dans le listener BellRingHandler1, affichez ‘Bellring handler 1’puis affichez ‘Tom aide-moi’ si [arg1] == ‘Jerry’, sinon affichez ‘Bonjour [arg1]

✓ Dans le listener BellRingHandler2, affichez ‘Bellring handler 2’puis exécutez un autre listener NobodyHandler. Ce dernier affiche ‘Desolé y a plus de personne ici laissez votre message svp’

- Ajouter les Listeners BellRing dont BellRingHandler1 et BellRingHandler2, et le listener NobodyHandler
- Testez (Emettez) l’événement avec une seul argument ‘Jerry’

Voici le résultat :

```
Bellring handler 1

Tom aide-moi

Bellring handler 2

Desolé y a plus de personne ici laissez votre message svp
```