MyNodeJS

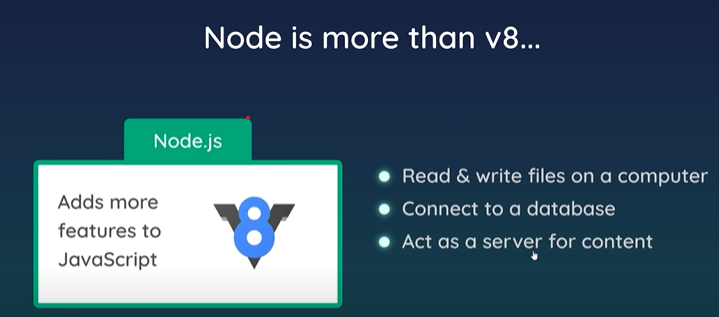
1. Getting started
2. Introduction

Node.js est une plateforme logicielle open-source basée sur le moteur JavaScript V8 de Google(**V8 aussi nous aide a compiler le code Js sur le navigateur**). Elle permet l'exécution de code JavaScript côté serveur, permettant ainsi de développer des applications web et des services réseau performants et évolutifs.

Une image contenant diagramme

Description générée automatiquement

V8 nous aide a run le code Js sur notre serveur ou notre computer



Node.js a un avantage unique car des millions de développeurs frontaux qui écrivent du JavaScript pour le navigateur sont maintenant capables d'écrire le code côté serveur en plus du code côté client sans avoir à apprendre un langage complètement différent.

Dans Node.js, les nouvelles normes ECMAScript peuvent être utilisées sans problème, car vous n'avez pas besoin d'attendre que tous vos utilisateurs mettent à jour leurs navigateurs - vous avez la charge de décider quelle version ECMAScript utiliser en changeant la version de Node.js, et vous pouvez également activer des fonctionnalités expérimentales spécifiques en exécutant Node.js avec des drapeaux

**Un exemple d'application Node.js**

const http = require('http');

const hostname = '127.0.0.1';

const port = 3001;

const server = http.createServer((req, res) => {

  res.statusCode = 200;

  res.setHeader('Content-Type', 'text/plain');

  res.end('Hello World\n');

});

server.listen(port, hostname, () => {

  console.log(`Server running at http://${hostname}:${port}/`);

});

Pour exécuter ce code, sauvegardez-le en tant que fichier server.js et exécutez node server.js dans votre terminal.

Ce code inclut d'abord le [module http] de Node.js (<https://nodejs.org/api/http.html>).

Node.js possède une fantastique [bibliothèque standard](https://nodejs.org/api/), y compris un support de première classe pour les réseaux.

La méthode createServer() de http crée un nouveau serveur HTTP et le renvoie.

Le serveur est configuré pour écouter sur le port et le nom d'hôte spécifiés. Lorsque le serveur est prêt, la fonction de rappel est appelée, dans ce cas, pour nous informer que le serveur est en cours d'exécution.

Lorsqu'une nouvelle requête est reçue, l'événement [request](https://nodejs.org/api/http.html" \l "http_event_request) est appelé, fournissant deux objets : une requête (un objet [http.IncomingMessage](https://nodejs.org/api/http.html" \l "http_class_http_incomingmessage)) et une réponse (un objet [http.ServerResponse](https://nodejs.org/api/http.html" \l "http_class_http_serverresponse)).

Ces 2 objets sont essentiels pour gérer l'appel HTTP.

Le premier fournit les détails de la requête. Dans cet exemple simple, il n'est pas utilisé, mais vous pouvez accéder aux en-têtes et aux données de la requête.

Le second est utilisé pour retourner les données à l'appelant.

Dans ce cas, avec :

js

copy

res.statusCode = 200;

Nous définissons la propriété statusCode à 200, pour indiquer une réponse réussie.

Nous définissons l'en-tête Content-Type :

js

copy

res.setHeader('Content-Type', 'text/plain');

et nous fermons la réponse, en ajoutant le contenu comme argument à end() :

js

copy

res.end('Hello World\n');

1. Combien de Js devez nous connaitre pour utiliser NodeJs

Une image contenant texte

Description générée automatiquement

Une image contenant texte

Description générée automatiquement

1. Le moteur js V8

Une image contenant texte

Description générée automatiquement

1. Npm

C’est quoi npm

Un guide rapide de npm, le puissant gestionnaire de paquets clé du succès de Node.js. En janvier 2017, plus de 350000 paquets ont été signalés comme étant répertoriés dans le registre npm, ce qui en fait le plus grand dépôt de code monolingue sur Terre, et vous pouvez être sûr qu'il y a un paquet pour (presque !) tout.

## Paquets

**Une image contenant texte, capture d’écran, écran, noir

Description générée automatiquement**Une image contenant texte, capture d’écran, écran, sombre

Description générée automatiquement

**Une image contenant texte

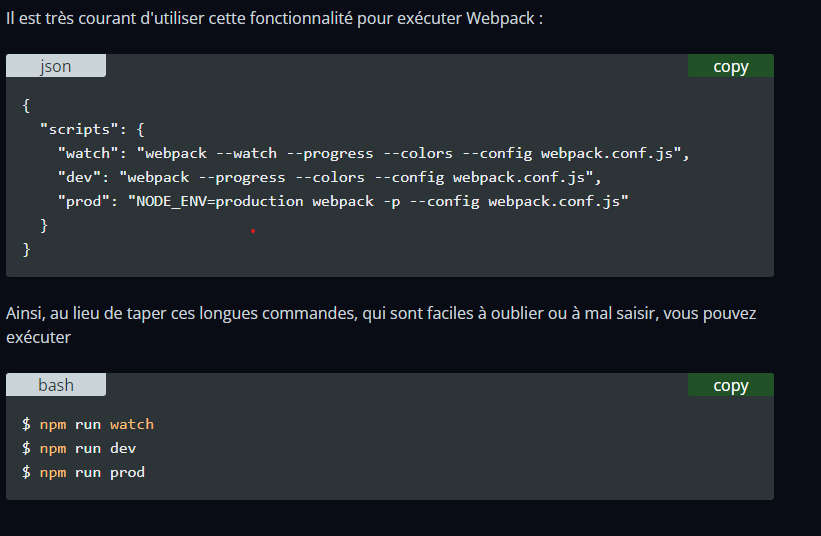
Description générée automatiquement**

**Une image contenant texte

Description générée automatiquement**

**Une image contenant texte, capture d’écran, écran, intérieur

Description générée automatiquement**

****

1. **La difference entre le développement et la production**

Une image contenant texte

Description générée automatiquement

**Une image contenant texte

Description générée automatiquement**

1. **NodeJs Avec Typescript**

Node.js est souvent utilisé en combinaison avec TypeScript pour améliorer la robustesse et la sécurité des applications back-end. L'utilisation de TypeScript avec Node.js peut aider à prévenir les erreurs de type courantes qui peuvent survenir lors du développement d'applications JavaScript de grande envergure.

**Une image contenant texte, capture d’écran, noir, écran

Description générée automatiquement** **Une image contenant texte

Description générée automatiquement**

1. **nodeJs et WebAssembly**

[**https://nodejs.dev/fr/learn/nodejs-with-webassembly/**](https://nodejs.dev/fr/learn/nodejs-with-webassembly/)

1. Asynchrone work
2. **Deffinition des callbacks ,asynchrone ,synchrone**

**JavaScript est synchrone par défaut et est un single thread. Cela signifie que le code ne peut pas créer de nouveaux threads et s'exécuter en parallèle.**

* 1. **Les callbacks**

Une fonction callback, ou simplement un rappel en français, est une fonction qui est passée en tant qu'argument à une autre fonction et qui est exécutée après l'exécution de cette fonction. La fonction callback est généralement utilisée pour exécuter du code après une opération asynchrone, telle que la lecture d'un fichier ou l'appel d'une API réseau.

La fonction callback est appelée une fois que l'opération asynchrone est terminée et que les résultats sont prêts à être traités. Elle permet donc de traiter des données de manière asynchrone, sans bloquer l'exécution du programme. La fonction callback peut également prendre des arguments pour recevoir des données ou des erreurs retournées par l'opération asynchrone.

Les fonctions callback sont largement utilisées en JavaScript et sont une partie importante de la programmation asynchrone. Les bibliothèques et les frameworks JavaScript modernes, tels que Node.js, jQuery et React, utilisent souvent des fonctions callback pour gérer les événements et les opérations asynchrones.

Voici un exemple simple de fonction callback en JavaScript :

function direBonjour(nom, callback) {

  const message = `Bonjour, ${nom}!`;

  callback(message);

}

function afficherMessage(message) {

  console.log(message);

}

direBonjour("Alice", afficherMessage); // affiche "Bonjour, Alice!"

Dans cet exemple, la fonction **direBonjour** prend un nom et une fonction callback **callback** en arguments. Elle construit ensuite un message personnalisé en utilisant le nom, et appelle la fonction callback en lui passant ce message en argument. La fonction **afficherMessage** est passée comme callback, elle se contente d'afficher le message sur la console.

* 1. **Les fonctions asynchrone**

Une fonction asynchrone, également connue sous le nom de fonction asynchrone, est une fonction qui permet l'exécution de code de manière asynchrone. Contrairement à une fonction synchrone, qui exécute du code de manière séquentielle et bloque l'exécution du programme jusqu'à la fin de l'opération en cours, une fonction asynchrone permet à d'autres opérations de s'exécuter en parallèle.

Lorsqu'une fonction asynchrone est appelée, elle est exécutée de manière asynchrone, c'est-à-dire que le programme ne bloque pas l'exécution du code et continue à s'exécuter en parallèle. La fonction asynchrone peut utiliser des fonctions callback, des promesses ou des mots-clés tels que **async** et **await** pour gérer les opérations asynchrones.

L'utilisation de fonctions asynchrones est très courante en programmation JavaScript, en particulier dans le contexte du développement web. Les opérations asynchrones telles que les appels API, les lectures et écritures de fichiers et les requêtes de bases de données peuvent toutes être gérées de manière asynchrone à l'aide de fonctions asynchrones.

Voici un exemple de fonction asynchrone en utilisant les mots-clés **async** et **await**

async function maFonctionAsynchrone() {

  const resultat = await faireQuelqueChoseAsynchrone();

  console.log(resultat);

}

async function faireQuelqueChoseAsynchrone() {

  return new Promise((resolve) => {

    setTimeout(() => {

      resolve("Résultat asynchrone");

    }, 1000);

  });

}

maFonctionAsynchrone();

Dans cet exemple, la fonction **faireQuelqueChoseAsynchrone** renvoie une promesse qui est résolue avec le résultat "Résultat asynchrone" après une seconde d'attente. La fonction **maFonctionAsynchrone** est une fonction asynchrone qui utilise le mot-clé **await** pour attendre la résolution de la promesse renvoyée par **faireQuelqueChoseAsynchrone**, puis affiche le résultat sur la console.

* 1. **Une fonction synchrone**

Une fonction synchrone, ou simplement une fonction, est une fonction qui exécute du code de manière synchrone, c'est-à-dire qu'elle exécute chaque instruction dans l'ordre dans lequel elle est écrite, de manière séquentielle.

1. **Contrôle de flux asynchrone**

Le contrôle de flux asynchrone fait référence à la manière dont les opérations asynchrones sont gérées dans un programme. Les opérations asynchrones peuvent prendre plus de temps pour être complétées que les opérations synchrones, et elles sont généralement exécutées en arrière-plan ou sur un autre thread pour éviter de bloquer le thread principal.

Le contrôle de flux asynchrone est utilisé pour gérer les résultats de ces opérations asynchrones de manière efficace et éviter les blocages de programme ou les erreurs de traitement. Les techniques courantes de contrôle de flux asynchrone incluent l'utilisation de callbacks, de promesses, de générateurs et d'async/await.

Les callbacks sont utilisés pour exécuter une fonction de rappel après la fin d'une opération asynchrone. Cependant, ils peuvent rapidement devenir difficiles à lire et à gérer lorsqu'il y a plusieurs opérations asynchrones imbriquées.

Les promesses sont une alternative aux callbacks qui offrent une syntaxe plus lisible pour gérer les opérations asynchrones. Les promesses sont des objets qui représentent une valeur qui peut être résolue de manière asynchrone. Les promesses peuvent être chaînées et combinées pour gérer les résultats de plusieurs opérations asynchrones.

Les générateurs sont une autre technique de contrôle de flux asynchrone qui permet de suspendre et de reprendre l'exécution d'une fonction en cours d'exécution. Les générateurs sont souvent utilisés avec des promesses pour créer des fonctions asynchrones plus lisibles et plus faciles à gérer.

Enfin, l'utilisation d'async/await est une méthode de contrôle de flux asynchrone introduite dans ECMAScript 2017. Cette technique permet de gérer les opérations asynchrones en utilisant une syntaxe plus proche de la programmation synchrone, en utilisant les mots-clés **async** et **await**.

En conclusion, le contrôle de flux asynchrone est une technique importante pour gérer les opérations asynchrones dans un programme. Les callbacks, les promesses, les générateurs et l'async/await sont des méthodes courantes de contrôle de flux asynchrone utilisées dans les langages de programmation modernes.

* 1. **Flux de contrôle**

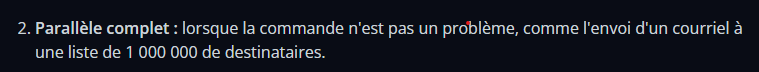
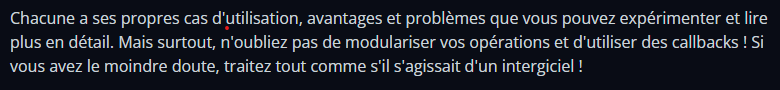
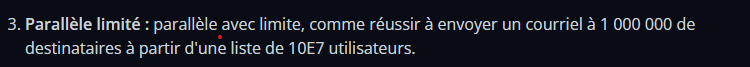
Une image contenant texte

Description générée automatiquement

Une image contenant texte

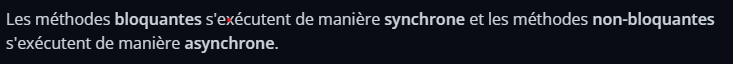
Description générée automatiquement

**Une image contenant texte

Description générée automatiquement** **** 

* 1. **Vue d’ensemble du blocage et du non-blocage**

Cette présentation couvre la différence entre les appels bloquants et non-bloquants dans Node.js. Cet aperçu fera référence à la boucle d'événement et à libuv mais aucune connaissance préalable de ces sujets n'est requise**.**

****

**Une image contenant texte

Description générée automatiquement**

Une image contenant texte

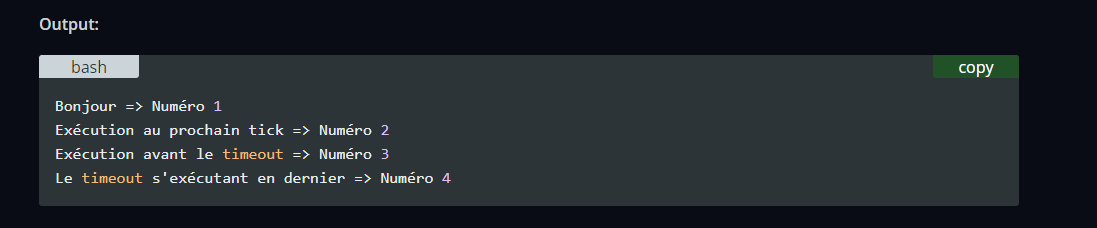
Description générée automatiquement

En Node.js, **process.nextTick()** est une méthode qui permet de planifier l'exécution d'une fonction pour le prochain cycle d'événement du processus Node.js. Contrairement à la méthode **setImmediate()**, qui planifie l'exécution d'une fonction pour le prochain cycle d'événement du bouclage d'événements (event loop), **process.nextTick()** planifie l'exécution d'une fonction pour le cycle d'événement suivant du processus.

L'utilisation de **process.nextTick()** permet de reporter l'exécution d'une fonction jusqu'à ce que toutes les opérations synchrones en cours soient terminées, ce qui peut améliorer les performances et éviter les blocages de boucle d'événements. Cela peut également aider à éviter les problèmes de synchronisation liés à l'utilisation de callbacks et de promesses.

En résumé, **process.nextTick()** est une méthode utile pour la programmation asynchrone en Node.js, car elle permet de planifier l'exécution d'une fonction pour le prochain cycle d'événement du processus, ce qui peut améliorer les performances et éviter les blocages.

Une image contenant texte

Description générée automatiquement

1. Les minuteur Js

Une image contenant texte

Description générée automatiquementUne image contenant texte

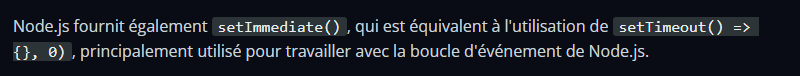
Description générée automatiquementUne image contenant texte, capture d’écran, écran

Description générée automatiquementUne image contenant texte, capture d’écran, écran, noir

Description générée automatiquementUne image contenant texte, capture d’écran, écran, fermer

Description générée automatiquementUne image contenant texte, capture d’écran, écran

Description générée automatiquementUne image contenant texte, capture d’écran, écran, noir

Description générée automatiquement

1. Comprendre setImmediate

Une image contenant texte, capture d’écran, écran

Description générée automatiquement

Une image contenant texte, capture d’écran, écran, noir

Description générée automatiquement

1. L’émetteur d’évènements de nodeJs

Une image contenant texte, capture d’écran, noir, écran

Description générée automatiquementUne image contenant texte, capture d’écran, écran

Description générée automatiquement

1. Manipulation Files
2. Statistoque des fichiers node.js

En Node.js, le module **fs** (pour "file system") fournit des méthodes pour effectuer des opérations de lecture et d'écriture de fichiers et de répertoires sur le système de fichiers du système d'exploitation. La méthode **fs.stat()** est l'une de ces méthodes et permet de récupérer des informations sur un fichier ou un répertoire.

La méthode **fs.stat()** prend en paramètre un chemin vers un fichier ou un répertoire et un callback qui sera appelé avec deux arguments : une erreur en cas d'échec de l'opération, et un objet qui contient des informations sur le fichier ou le répertoire.

Les informations contenues dans l'objet dépendent du type de fichier ou de répertoire. Voici quelques-unes des informations les plus courantes :

* **dev**: le numéro de périphérique du fichier ou du répertoire
* **ino**: le numéro d'i-noeud (inode) du fichier ou du répertoire
* **mode**: les permissions de fichier (dans un format octal)
* isFile: true ou false
* **isDirectory**: true ou false
* isSymbolicLink: les permissions de fichier (dans un format octal)
* **size**: la taille du fichier en octets
* **mtime**: la date de modification du fichier (un objet **Date**)

La méthode **fs.stat()** est utile pour vérifier si un fichier ou un répertoire existe, pour récupérer des informations sur un fichier (par exemple, sa taille ou sa date de modification), ou pour déterminer s'il s'agit d'un fichier ou d'un répertoire.

Une image contenant texte

Description générée automatiquement

1. Chemins d’aces aux fichiers NodeJs
   1. Extraire des informations d’un chemin

Une image contenant texte

Description générée automatiquement

* 1. Travailler avec des chemins

Une image contenant texte

Description générée automatiquement

1. Travailler avec des descripteurs de fichiers dans NodeJS

En Node.js, les descripteurs de fichiers sont des identificateurs numériques qui représentent des fichiers ouverts par le programme. Ils permettent de manipuler un fichier ouvert en effectuant des opérations telles que la lecture, l'écriture, la fermeture ou la modification des options de fichier.

Pour ouvrir un fichier en mode lecture, écriture ou lecture/écriture, vous pouvez utiliser la méthode **fs.open()** qui retourne un descripteur de fichier. Par exemple, pour ouvrir un fichier en mode écriture, vous pouvez utiliser le code suivant :

fs.open('monfichier.txt', 'w', (err, fd) => {

  if (err) throw err;

  console.log('Fichier ouvert avec succès');

  // Manipulation du fichier avec le descripteur de fichier fd

});

Dans cet exemple, la méthode **fs.open()** prend en premier paramètre le chemin d'accès au fichier et en deuxième paramètre le mode d'ouverture (**'r'** pour la lecture, **'w'** pour l'écriture ou **'r+'** pour la lecture/écriture). La méthode prend également un callback qui sera appelé avec deux arguments : une erreur en cas d'échec de l'opération et un descripteur de fichier (**fd**) en cas de succès.

Une fois que vous avez un descripteur de fichier, vous pouvez utiliser d'autres méthodes **fs** telles que **fs.read()**, **fs.write()** ou **fs.close()** pour manipuler le fichier. Par exemple, pour écrire dans un fichier à l'aide d'un descripteur de fichier, vous pouvez utiliser le code suivant :

const fs = require('fs');

fs.open('monfichier.txt', 'w', (err, fd) => {

  if (err) throw err;

  const buffer = Buffer.from('Contenu à écrire dans le fichier');

  const offset = 0;

  const length = buffer.length;

  const position = null; // Écrit à partir de la fin du fichier

  fs.write(fd, buffer, offset, length, position, (err, bytesWritten, buffer) => {

    if (err) throw err;

    console.log(`${bytesWritten} octets écrits`);

  });

});

Dans cet exemple, la méthode **fs.write()** est utilisée pour écrire dans le fichier à l'aide du descripteur de fichier (**fd**). Le premier paramètre est le descripteur de fichier, le deuxième paramètre est le contenu à écrire sous forme de **Buffer**, le troisième paramètre est l'offset dans le **Buffer** à partir duquel écrire, le quatrième paramètre est la longueur des données à écrire et le cinquième paramètre est la position de départ pour écrire les données dans le fichier. Dans cet exemple, la position est définie sur **null** pour écrire à partir de la fin du fichier.

Une image contenant texte

Description générée automatiquement

1. Lire des fichiers avec NodeJs

Une image contenant texte, capture d’écran, écran, noir

Description générée automatiquement

1. Ecrire des fichiers dans nodeJS

Une image contenant texte

Description générée automatiquement

Une image contenant texte, écran, capture d’écran, noir

Description générée automatiquementUne image contenant texte, capture d’écran, écran

Description générée automatiquement

1. La différence entre fs.open et fs.readFile and fs.writeFile

La principale différence entre **fs.readFile()** et **fs.open()** en Node.js est la façon dont ils manipulent les fichiers.

**fs.readFile()** est utilisé pour lire le contenu d'un fichier en entier de manière asynchrone, et renvoie le contenu sous forme de chaîne de caractères ou de tampon (**Buffer**). Cette méthode est pratique pour lire rapidement le contenu d'un fichier, mais elle n'offre pas autant de contrôle sur la manière dont le fichier est lu.

**fs.open()** est utilisé pour ouvrir un fichier en mode lecture, écriture ou lecture/écriture, et renvoie un descripteur de fichier qui peut ensuite être utilisé pour manipuler le fichier en effectuant des opérations telles que la lecture, l'écriture ou la fermeture du fichier. Cette méthode offre un contrôle plus fin sur la manière dont le fichier est manipulé, mais elle nécessite plus de code pour être utilisée.

En résumé, **fs.readFile()** est pratique pour lire rapidement le contenu d'un fichier, tandis que **fs.open()** offre un contrôle plus fin sur la manipulation du fichier. Le choix entre ces deux méthodes dépendra des besoins spécifiques de votre programme.

1. Travailler avec des dossiers dans NodeJs

Une image contenant texte

Description générée automatiquement

Une image contenant texte, écran, capture d’écran, téléphone cellulaire

Description générée automatiquement

Une image contenant texte, écran, capture d’écran, téléphone

Description générée automatiquement

Une image contenant texte, capture d’écran, écran, argent

Description générée automatiquement