





1 - Intro



Definition

Environnement d'exécution pour Javascript en dehors du navigateur

Open Source, multi-plateforme, très rapide

Communication avec un serveur

Javascript Everywhere (front et back)

Librairie open-source npm (node package manager)

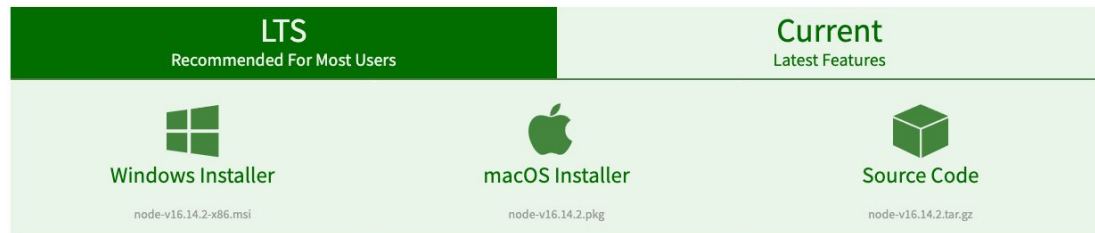
installation



Downloads

Latest LTS Version: **16.14.2** (includes npm 8.5.0)

Download the Node.js source code or a pre-built installer for your platform, and start developing today.



Windows Installer (.msi)

Windows Binary (.zip)

macOS Installer (.pkg)

32-bit	64-bit
32-bit	64-bit
64-bit / ARM64	

2 - utilisation REPL

Read Evaluate Print Loop

Vous écrivez du code et node
évalue votre code
directement dans le terminal

Depuis le terminal

commande node

```
[> const prenom= "jean"  
undefined  
[> prenom  
'jean'  
> █
```

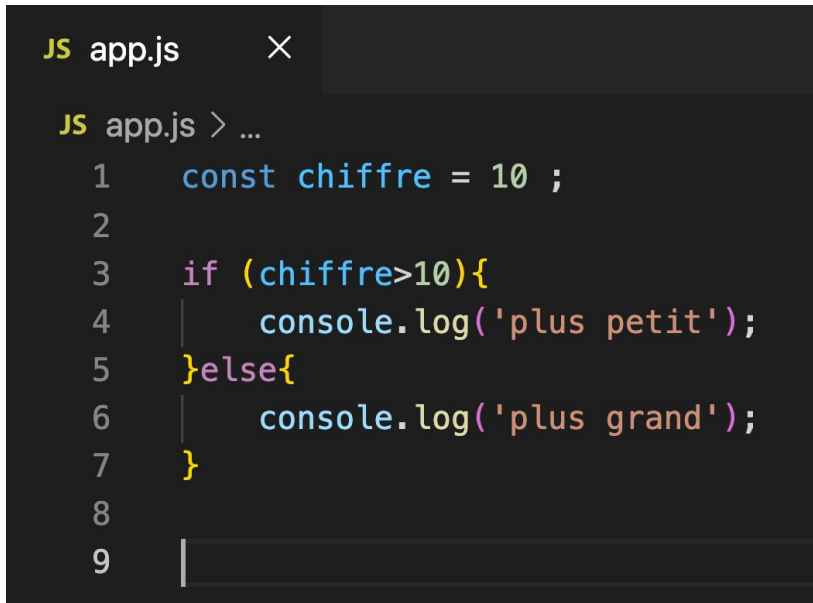
3 - utilisation CLI Command Line Interface

créer un dossier node
créer un fichier app.js

dans le terminal:

\$ node app.js

Besoin de réécrire la commande
après chaque modification par
défaut.



```
JS app.js ×  
JS app.js > ...  
1  const chiffre = 10 ;  
2  
3  if (chiffre>10){  
4      console.log('plus petit');  
5  }else{  
6      console.log('plus grand');  
7  }  
8  
9  |
```

ouverture du terminal depuis VSCode

Control + `

Terminal > New terminal

View > Terminal

variables globales

principe : variables accessibles depuis n'importe quel endroit dans l'application

```
// GLOBALS - pas l'objet Window (pas de navigateur)

// __dirname - chemin du dossier actuel
// __filename - nom du fichier
// require - fonction pour utiliser un module (commonJS)
// module - info sur le module actuel (file)
// process - info sur l'environnement où le programme est exécuté
```


Organisation en modules

encapsulated code : concept CommonJS. Chaque fichier est un module par défaut.

On ne partage que le minimum.

besoin d'import/export pour utiliser des variables ou fonctions contenues dans un autre fichier

app.js

```
JS app.js > ...
```

```
1
2   const pierre = 'pierre';
3   const julie = 'julie';
4
5   const direBonjour = (name) => {
6     |     console.log(`Hello ${name}`);
7   }
8
9   direBonjour(pierre);
10  direBonjour(julie);
```

Tout est rassemblé dans un seul et même fichier.

=> possibilité d'organiser son code en créant d'autres fichiers

fichier noms.js

```
JS 4-noms.js > ...
```

```
1  const pierre = 'pierre';  
2  const julie = 'julie';  
3  
4  console.log(module);  
5  //module.exports = {pierre: pierre, julie: julie};  
6  module.exports = {pierre, julie};  
7
```

On choisit précisément quels éléments sont accessibles depuis un autre fichier avec l'objet `module.exports`

fichier fonctions.js

JS 5-fonctions.js > ...

```
1  const direBonjour = (name) => {  
2    |    console.log(`Hello ${name}`);  
3  }  
4  
5  module.exports = direBonjour;  
...
```

dans app.js, require() pour accéder aux fichiers externes

```
// Modules
const noms = require('./4-noms');
// ou const {pierre, julie} => technique du destructuring
const direBonjour = require('./5-fonctions');

direBonjour(noms.pierre);
direBonjour(noms.julie);
```

On fait appel aux fichiers externes avec la fonction require(). require() accepte un chemin d'accès comme paramètre.

Attention

Si une variable n'est pas exportée, elle ne peut être importée.

```
const pierre = 'pierre';  
const julie = 'julie';  
const prive = "privé";  
  
console.log(module);  
//module.exports = {pierre: pierre, julie: julie};  
module.exports = {pierre, julie};
```

```
direBonjour(noms.pierre);  
direBonjour(noms.julie);  
direBonjour(noms.prive); // undefined
```

module = fonction

```
JS 7-autoExe > ...
```

```
1  const num1 = 10;
2  const num2 = 20;
3
4  function add(){
5      console.log(`${num1 + num2}`);
6  }
7
8  add();
9
```

```
// dans app.js
require('./7-autoExe');
```

Un fichier est une fonction.
Si une fonction est présente dans un fichier importée, elle sera déclenchée par le fichier qui l'importe.

4 - Built-in fonctions/ modules

Un très grand nombre de fonctions/modules est disponible pour réaliser toutes sortes de fonctionnalités.

<https://nodejs.org/dist/latest-v14.x/docs/api/documentation.html>

Node.js

About this documentation

Usage and example

Assertion testing

Async hooks

Buffer

C++ addons

C/C++ addons with Node-API

C++ embedder API

Node.js v14.18.1 documentation

[Index](#) | [View on single page](#) | [View as JSON](#) | [View another version ▼](#) | [Edit on GitHub](#)

▼ Table of contents

- About this documentation
 - Contributing
 - Stability index
 - Stability overview
 - JSON output
 - System calls and man pages

demo avec le module os (operating system)

```
const os = require('os');  
// utilisation d'une méthode
```

```
os.
```

- [EOL] EOL const EOL: string
- arch
- { } constants
- cpus
- endianness
- freemem
- getPriority
- homedir

https://nodejs.org/dist/latest-v14.x/docs/api/os.html#os_os

utilisation de méthodes disponibles

```
// information sur l'utilisateur  
const user = os.userInfo();  
console.log(user)
```

```
console.log(`système est en fonctionnement depuis ${os.uptime()} secondes`);
```

```
{  
  uid: 501,  
  gid: 20,  
  username: 'macbookpro',  
  homedir: '/Users/macbookpro',  
  shell: '/bin/zsh'  
}
```

module path

```
const path = require('path');

console.log(path.sep); // returns /

const filePath = path.join('/contenu', 'sous-dossier', 'info.txt');
console.log(filePath, 'filePath');

const base = path.basename(filePath);
console.log(base);

const absolute = path.resolve(__dirname, 'contenu', 'sous-dossier', 'info.txt');
console.log(absolute, 'absolute');
```

filePath :
reconstitue un
chemin

Extrait le nom du
fichier d'un chemin

Donne le chemin en
absolu

File System - Lecture de fichiers - version synchrone

- utilisation de `FileSystem(fs)` pour accéder aux contenus d'un fichier.

Créer un dossier contenu + deux fichiers txt avec du texte

`readFileSync(path, options)`

```
// Synchronous = Bloquant
const { readFileSync, writeFileSync } = require ('fs');
console.log('start');

// read
// 2 params:  file path+ encoding utf-8
const first = readFileSync('./contenu/first.txt', 'utf-8');
const second = readFileSync('./contenu/second.txt', 'utf-8');
```

File System - Ecriture de fichiers

- utilisation de `FileSystem(fs)` pour écrire du contenu **(en écrasant)** dans un fichier

Créer deux fichiers txt avec du contenu

`writeFileSync(path, options)`

```
// create a new file with values
// 2 params: File name, value
writeFileSync('./contenu/result-sync.txt', `Voici le résultat ${first}, ${second}`)
```

File System - Ecriture de fichiers

- utilisation de FileSystem(fs)
pour ajouter du contenu dans un fichier.

Créer deux fichiers txt avec du contenu

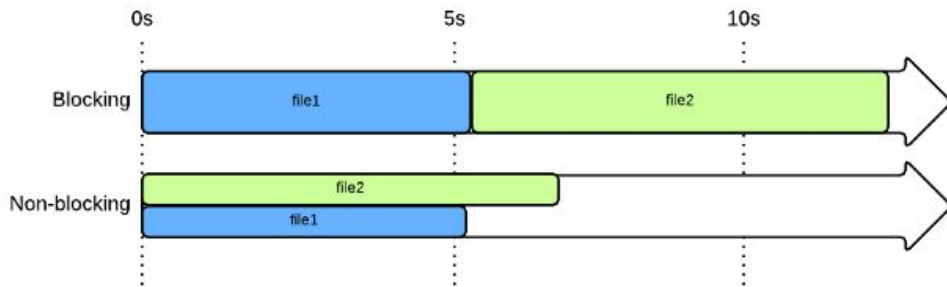
writeFileSync(path, options, flag)

```
// append to a file
// 3 params : file name, value, {flag:'a'}
writeFileSync(
  './contenu/result-sync.txt',
  `Le résultat : ${first}, ${second}`,
  { flag: 'a' }
);
```

sync/async

Utilisation de callback functions.

Permet au serveur de continuer à fonctionner en mettant certaines tâches en tâche de fond.



File System - version asynchrone

readFile et writeFile sont des fonctions avec callback.

<https://nodejs.dev/learn/reading-files-with-node>

Attention : ne pas oublier l'encodage utf-8 !

```
// asynchronous = non-bloquant;
// callback
const { readFile, writeFile } = require('fs');
console.log('start');

readFile('./contenu/first.txt', 'utf-8', (err, result) => {
  if(err){
    console.log(err)
    return
  }
  console.log(result, "result");
  const first = result;

  readFile('./contenu/second.txt', 'utf-8', (err, result) => {
    if(err){
      console.log(err);
      return
    }
    const second = result;
    console.log(second, 'second');

    writeFile(
      './contenu/result-async.txt',
      `Here is the result : ${first}, ${second}`,
      (err, result) => {
        if(err){
          console.log(err)
          return
        }
        console.log('done with this task')
      }
    )
  })
})

console.log('juste après la tâche');
```


module http

http: hypertext transfer protocol

```
const http = require ('http');

const server = http.createServer((req, res)=>{
  res.write('Hello World');
  // end the request
  res.end()
})

// define a server
server.listen(5000);
```

req et res sont des objets avec un ensemble de méthodes.
req => request ; ce qui a été demandé par l'utilisateur
res=> réponse du serveur

module http : définir des scénarios (routes provisoires)

```
const server = http.createServer((req, res)=>{  
  // Si route est homepage  
  if(req.url === "/"){  
    res.write('Hello World');  
    // end the request  
    res.end()  
  }  
  // Si route est /contact  
  if(req.url === "/contact"){  
    res.write('contact');  
    // end the request  
    res.end()  
  }  
  // autre route – réponse par défaut  
  res.end('oops');  
})
```

On définit des routes et des affichages associés.

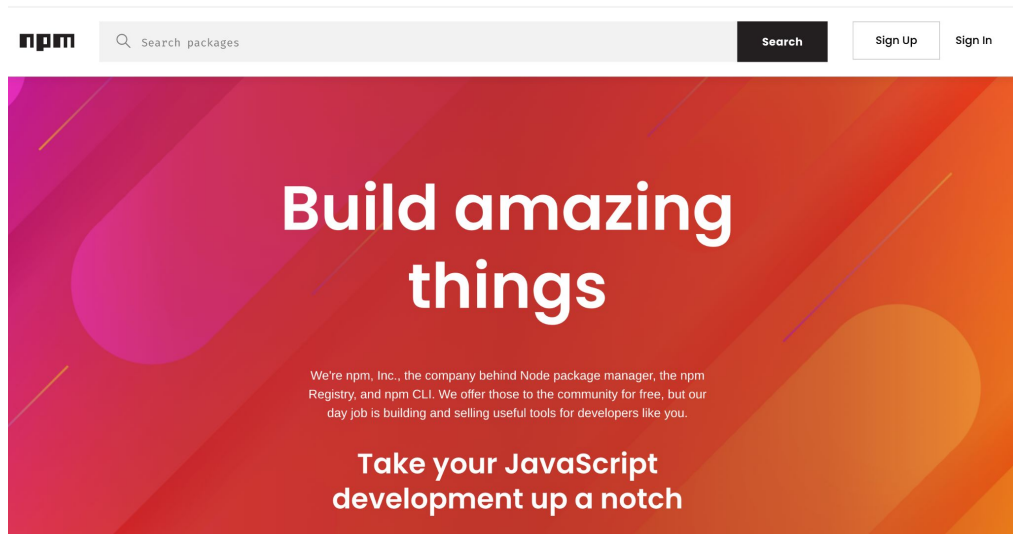
librairie npm - node package manager

Ensemble de code disponible avec node.

Dans l'univers node, le code réutilisable est un package (ou dependencies/module).

Node vous permet de réutiliser du code écrit par d'autres.

<https://www.npmjs.com/>



warning ! pas de contrôle de qualité dans npm. Travail du développeur de choisir et sélectionner le meilleur package.

Besoin de créer package.json

Ce fichier liste tous les packages installés dans l'application.

nom + version.

Ce fichier doit se situer à la racine du site.

Pour le créer :

- manuellement (peu utilisé)
- npm init (étapes à valider)
- npm init -y (ensemble de propriétés par défaut, y équivaut à yes)

CLI - command line interface

```
// npm node package manager
// npm --version

// installer au niveau local
// npm i <packagename>

// installer au niveau global
// npm i -g <packagename>
// sudo npm i <packagename> (mac)
```

En installant node, on a accès aux commandes npm

commandes nécessaires pour installer un package

Warning : besoin de créer package.json avant !

1/2 - installation de packages

```
npm install lodash
```

L'installation d'un package ajoute un dossier `node_modules/<nomdupackage>`
ajoute une propriété `dependencies` dans `package.json`.

Possibilité de supprimer le package
directement depuis VSCode ou en ligne de
commande.

2/2 Utilisation dans l'application

```
const _ = require('lodash')  
  
const items = [1, [2, [3, [4]]]]  
const newItems = _.flattenDeep(items)  
console.log(newItems)
```

partager son code (sur git)

créer un repo sur github

.gitignore (ignoré par search control)

git init

git remote add origin git@github.com:<votrecompte>/<votrerepo>.git

seul le fichier package.json est nécessaire pour retrouver tous les modules nécessaires.

Cloner un projet distant

Quand vous clonez un projet

```
git clone <url>
```

vous importez tout le dossier disponible sur github, sans le dossier node_modules.

Besoin de la commande **npm install**

Tous les packages spécifiés dans package.json seront importés comme dans le projet d'origine.

git rappel

les clés SSH se situent dans

Users/macbookpro/.ssh/

ssh -vT git@github.com pour connaître le nom du fichier avec la clé utilisée

nodemon est un utilitaire d'interface de ligne de commande (CLI) développé par @rem. Il enveloppe votre application Node, surveille le système de fichiers et redémarre automatiquement le processus

Il redémarre l'application dès qu'il détecte un changement.

<https://www.npmjs.com/package/nodemon>

ETAPES :

\$ npm install nodemon --save-dev (package dispo uniquement en dev)

ajouter un script dans package.json

exécuter le script avec \$npm run <nomduscript>

> Dans app.js

console.log('hello')

puis modifier pour ('hello world').

> nodemon surveille notre application.

nodemon



```
"scripts": {  
  "test": "echo \"Error: no test specified\" && exit 1",  
  "start": "node app.js",  
  "dev": "nodemon app.js"  
},
```

```
[nodemon] 2.0.14  
[nodemon] to restart at any time, enter `rs`  
[nodemon] watching path(s): *.*  
[nodemon] watching extensions: js,mjs,json  
[nodemon] starting `node app.js`  
[nodemon] clean exit - waiting for changes before restart
```

Pour arrêter nodemon

```
^ C (control + C)
```

Pour désinstaller un package

```
npm uninstall <package>
```

Autre option : supprimer le node_modules + package-lock

installer un package avec -g

Les dépendances sont installées par défaut dans le dossier depuis lequel a été fait la commande.

Parfois, besoin d'installer un package pour qu'il soit disponible dans tout le système, c'est-à-dire indépendamment de l'emplacement où il a été effectué.

```
$ npm i -g <nomDuPackage>
```

npm vs npx

npm : node package manager

npx : node package execute (package runner)

npx n'installe pas la dépendance dans votre système.

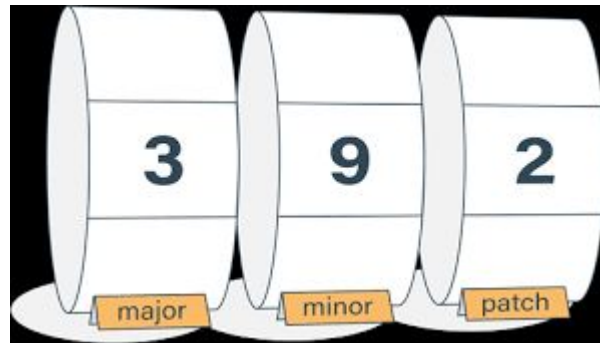
utile pour les commandes isolées, peu fréquentes, pour tester.

npx permet de ne pas fixer une version de package en particulier.

Source

<https://stackoverflow.com/questions/50605219/difference-between-npx-and-npm>

<https://nodesource.com/blog/an-absolute-beginners-guide-to-using-npm/>



Concepts clés avec node

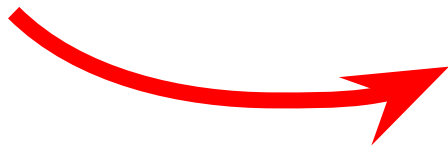
Event loop

Event loop - boucle d'événements

source:

<https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/>

NODE MET DE CÔTÉ (OFFLOAD) TOUT CE QUI PREND DU TEMPS,
AFIN DE CONTINUER À FAIRE FONCTIONNER L'APPLICATION



Jeter du lest



Dans l'univers JavaScript

- Javascript exécute le code de façon bloquante

```
// demo bloquant

console.log("premiere tâche");
for(i=0; i<10000; i++){
  console.log(i, "tout le monde m'attend");
}

console.log("deuxième tâche");
```

```
// demo non-bloquant
console.log("premiere tâche");
setTimeout(()=>{
  console.log("je prends du temps")
},5000);

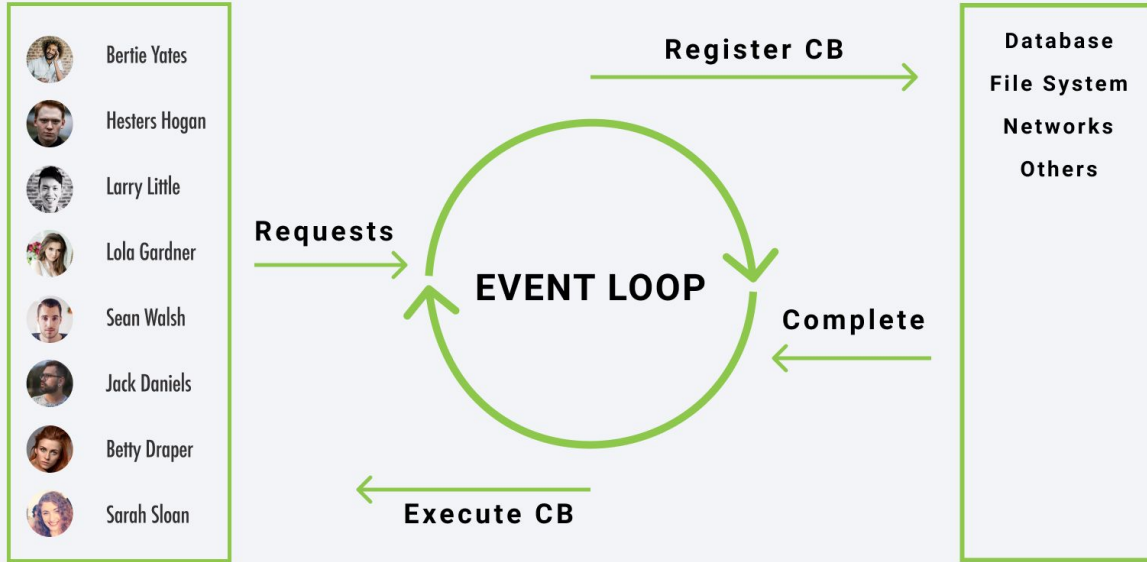
console.log("deuxième tâche");
```

setTimeout (ou fetch par exemple) permet de placer du code en tâche de fond. Même avec un timer à 0 (zéro) milliseconde, setTimeout s'exécute après tout le reste.

Event loop - boucle d'événements

Node.js fonctionne sur un seul fil d'exécution
mot-clé : Single Thread

La boucle est une boucle infinie.
Son travail est de traiter les événements, de les placer en fil d'attente



Event loop reçoit les requêtes utilisateurs.
Elle place les requêtes dans une file d'attente.
Une fois que ces requêtes sont exécutées, elle envoie le résultat côté client.

Dans ce cas, la boucle est exécutée immédiatement.

Ce code bloque tout le serveur en attendant son exécution.

```
const http = require('http');

const server = http.createServer((req, res) => {
  // Si route est homepage
  if (req.url === '/') {
    return res.end('bienvenue');
    // end the request
    // res.end()
  }
  // Si route est /contact
  if (req.url === '/contact') {
    for (let i = 0; i < 1000; i++) {
      for (j = 0; j < 1000; j++) {
        console.log(`${i} ${j}`);
      }
    }
    //res.write('contact');
    // end the request
    return res.end('contact');
  }
  // autre route - réponse par défaut
  res.end('page not found');
});

// define a server
server.listen(5000);
```

new Promise

alternative aux callbacks

Une promesse est un objet ([Promise](#)) qui représente la complétion ou l'échec d'une opération asynchrone.

https://developer.mozilla.org/fr/docs/Web/JavaScript/Guide/Using_promises

```
return new Promise (fonctionResolve, fonctionReject) =>{  
    // code à évaluer  
}
```

```
// 2 - refactor with new Promise
const getText = (path) => {
  return new Promise((resolve, reject) => {
    readFile(path, 'utf-8', (err, data) => {
      if (err) {
        reject(err);
      } else {
        resolve(data);
      }
    });
  });
};

// invoke getText return a promise (succès ou échec)
getText('./contenu/sous-dossier/info.txt')
  .then(result => console.log(result))
  .catch(err => console.log(err))

console.log('test');
```

The diagram illustrates the flow of data in the provided JavaScript code. A red box highlights the `reject(err);` statement inside the `readFile` callback. A red arrow originates from this box and points to the `.catch(err => console.log(err))` method call in the `getText` invocation. A green box highlights the `resolve(data);` statement inside the `readFile` callback. A green arrow originates from this box and points to the `.then(result => console.log(result))` method call in the `getText` invocation.

new Promise accepte deux arguments : resolve ou reject

On entoure la fonction `readFile` avec `new Promise`.

De cette façon, nous avons accès à `resolve` ou `reject` pour définir des scénarios (succès, échec).

On fait appel à la fonction `getText()`.

On a accès aux méthodes `.then` et `.catch`

async/await

alternative aux callbacks

```
// avec async await
const start = async() =>{
  try{
    const first = await getText('./contenu/first.txt');
    const second = await getText('./contenu/second.txt');
    console.log(first, second)
  }catch(err){
    console.log(err);
  }
}

start();
```

Gérer des événements

- Avec JavaScript => clic, scroll, timing
- Avec nodejs => également des événements existent
- Un signal qui indique que quelque chose a eu lieu
- exemple http : déclenche des événements

créer des événements - importer la fonctionnalité

```
// class
const EventEmitter = require('events');
// object (instance of our class)
const emitter = new EventEmitter();
```

créer des événements

ÉVÉNEMENT EST DÉCLENCHÉ

```
// .emit (déclencher)  
emitter.emit('event');
```

ÉVÉNEMENT EST DÉTECTÉ

```
// emit() envoie à tous les listeners  
// pour cela, il faut que les listeners soient déjà connus  
emitter.on('event', function(){  
  console.log('test')  
});
```

ATTENTION ! le détecteur d'événement doit être déclaré avant le emit()

```
// emit() envoie à tous les listeners/ abonnés
// pour cela, il faut que les listeners soient déjà connus
emitter.on('evenement', function(){
  console.log('test')
})
emitter.on('evenement', function () {
  console.log('autre conséquence');
});

// emitter avec des méthodes
// .emit (make a noise)
emitter.emit('evenement');
```

On peut cumuler plusieurs abonnés.

Attention, l'ordre est important !

```
// emit() envoie à tous les listeners/ abonnés
// pour cela, il faut que les listeners soient déjà connus
emitter.on('evenement', function(arg1, arg2){
  |   console.log('test', arg1, arg2)
})

emitter.on('evenement', function () {
  |   console.log('autre conséquence');
});

// emitter avec des méthodes
// .emit (make a noise)
emitter.emit('evenement', 'arg1', 'arg2');
```

Possibilité de passer des arguments à la fonction emit.

Dans la fonction .on(), on peut les récupérer.

créer des événements

Possibilité de multiplier les détecteurs d'événements (à la différence de onclick en javascript).

Ordre d'écriture impacte sur le résultat.

Besoin de déclarer d'abord les détecteurs d'événements (.on) avant l'émission de l'événement.

La méthode .emit() s'exécute immédiatement et ne tient pas compte de détecteurs déclarés plus tard.

refactoriser la requête http

documentation :

Event: 'request'

Added in: v0.1.0

- `request` `<http.IncomingMessage>`
- `response` `<http.ServerResponse>`

Emitted each time there is a `request`. There may be multiple `request`s per connection (in the case of HTTP Keep-Alive connections).

refactoriser la requête http

Que remarque-t-on ?

=> pas besoin de déclarer l'événement 'request'. Il fait partie des fonctionnalités disponibles.

```
// const server = http.createServer((req, res)=>{  
//     // res.end('Bienvenue')  
// })
```

```
const server = http.createServer();  
  
server.on('request', (req, res)=>{  
    // Si route est homepage  
    if(req.url === "/"){  
        // res.write();  
        // end the request  
        res.end('hello world')  
    }  
    // Si route est /contact  
    if(req.url === "/contact"){  
        //res.write('contact');  
        // end the request  
        res.end('contact')  
    }  
    // autre route – réponse par défaut  
    //res.end('oops');
```

Streams (flux)

- stream/flux pour lire ou écrire de façon séquentielle.

Meilleure performance que lire ou écrire en bloc.

Permet de procéder par étape.
Pertinent quand fichier volumineux.

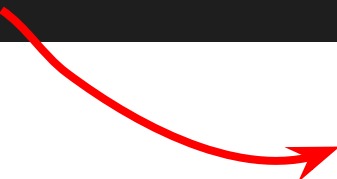
- *read*
- *write*
- *duplex (lire et écrire)*
- *transform*



createReadStream

Mise en pratique : création d'un fichier volumineux

```
JS createur-fichier-vol.js ×  
contenu > JS createur-fichier-vol.js > ...  
1  const { writeFileSync } = require('fs');  
2  
3  for (let i=0; i<10000; i++){  
4    writeFileSync("./contenu/fichier-vol.txt", `${i}`, {flag:'a'});  
5  }  
6
```



```
fichier-vol.txt — nodeJS-freecodecamp  
JS createur-fichier-vol.js  fichier-vol.txt ×  
contenu > ≡ fichier-vol.txt  
1  01234567891011121314151617181920212223242526272  
49505152535455565758596061626364656667686970717  
93949596979899100101102103104105106107108109110  
41251261271281291301311321331341351361371381391  
15415515615715815916016116216316416516616716816  
83184185186187188189190191192193194195196197198
```

createReadStream

```
JS app.js > ...
1 | // streams from fs( file system)
2 | const { createReadStream } = require('fs');
3 |
4 | const stream = createReadStream('./contenu/fichier-vol.txt');
5 |
6 | stream.on('data', (result) => {
7 |   console.log(result);
8 | })
9 | <Buffer 30 0a 31 0a 32 0a 33 0a 34 0a 35 0a 36 0a 37 0a 38 0a 39 0a 31 30 0a 31 31 0a 31 32 0a 31
   33 0a 31 34 0a 31 35 0a 31 36 0a 31 37 0a 31 38 0a 31 39 0a ... 48840 more bytes>
[nodemon] clean exit - waiting for changes before restart
```

l'événement 'data' est fourni par nodejs et nous donne un état des lieux des données lues (chunk).

par défaut un chunk = 64kb

<https://nodejs.org/api/stream.html#event-data>

createReadStream

par défaut chaque chunk est d'environ 65kb.

possibilité de modifier la taille du paquet avec highWaterMark

```
const stream = createReadStream('./contenu/fichier-vol.txt',  
  {  
    highWaterMark: 90000,  
    encoding: 'utf-8'  
  });
```

Le dernier paquet correspond aux données restantes.

```
0a ... 89950 more bytes>  
5 34 0a 33 31 38 35 35 0a 33  
0a ... 89950 more bytes>  
5 34 0a 34 36 38 35 35 0a 34  
0a ... 89950 more bytes>  
5 34 0a 36 31 38 35 35 0a 36  
0a ... 89950 more bytes>
```

createReadStream - gestion des erreurs

```
// écouter les erreurs
stream.on('error', (err)=>{
  console.log(err, 'erreur');
})
```


afficher en chunk (morceau)

problématique: afficher de nombreuses données de façon fluide.

Au préalable, on peut créer un fichier encore plus volumineux.

```
const http = require('http');
const fs = require('fs')

http.createServer((req, res)=>{
  const text = fs.readFileSync('./contenu/fichier-vol.txt', 'utf-8')
  res.end(text)
}).listen(5000)
```

	Size	Time	Waterfal
	1.2 MB	102 ms	
	1.2 MB	8 ms	
e.j...	451 B	35 ms	

× Headers Preview Response Initiator

Remote Address: [::1]:5000

Referrer Policy: strict-origin-when-cross-o

▼ Response Headers View source

Connection: keep-alive

Content-Length: 1177780

Date: Mon, 21 Mar 2022 15:23:41 GMT

Keep-Alive: timeout=5

createReadStream pour afficher en chunk (morceau)

problématique: afficher de nombreuses données de façon fluide.

```
http.createServer((req, res)=>{
  const fileStream = fs.createReadStream('./contenu/fichier-vol.txt', 'utf8');
  fileStream.on('open', ()=>{
    fileStream.pipe(res)
  })
  fileStream.on('error', (error)=>{
    res.end(error)
  })
}).listen(5000)
```

▼ Response Headers

[View source](#)

Connection: keep-alive

Date: Mon, 21 Mar 2022 15:35:02 GMT

Keep-Alive: timeout=5

Transfer-Encoding: chunked

La méthode .pipe() permet d'écrire par paquets.

<https://nodejs.org/en/knowledge/advanced/streams/how-to-use-stream-pipe/>