

PluralSight: Node.js

Node.js : The big picture

Considering Node.js:

Node.js => Asynchronous event driven Javascript runtime

Where is it commonly found?

You don't have a server where to upload your code. Your code is your server!



=> Microservices and API

↳ Node.js can be very simple and lightweight for it.

=> Serverless Cloud Functions (AWS, Azure, ...)

↳ Dynamic language runtime, fast startup and low memory consumption.

=> Command Line Applications (Webpack, Gulp, ESLint, Yeoman, ...)

=> Desktop Applications



↳ Framework Electron

=> Skype, Slack, Visual Studio code, GitHub Desktop, ...

What makes up Node.js?

→ V8 Javascript engine

→ Library => Event loop + Asynchronous I/O

→ Additional custom code

=> No more browser incompatibilities or polyfills!

node.green => site to see which versions of node.js

support what functionality

=> Need to make adjustments when changing versions.

=> You may need to be aware of the OS on which Node.js is running. -> other adjustments to make

New major version of Node.js every 6 months

When is Node.js not a good fit?

=> CPU-intensive tasks

=> When you don't like JavaScript

Thinking asynchronous.

Node.js event's loop

Different ways to manage multiple clients:

-> Process-per-client (Multi-Process)

-> Thread-per-client (Multi-Threaded)

-> Event Loop ("Single Threaded")

Node.js uses an event loop. All "actions" are taken one after another, nothing is running simultaneously.

Ls Everything is served by the same code, running in the same process, in the same loop.

=> There is no process or thread isolation.

Everything is linked to the event loop.

It only takes care of one task at a time.

You need to watch out that certain actions/treatments will not block the event loop.

The thread can't spend too much time on one task *

Asynchronous development

Exemple d'un restaurant : une serveuse fait l'aller-retour entre les clients et le chef qui prépare les repas.

Principe de la programmation asynchrone :

On délégue une action à un autre thread (noyau de l'OS hôte) que le thread unique de l'event loop. La suite de cette action est appelée par une fonction de callback quand cette action est terminée.

- Entre temps, le thread de l'event loop peut effectuer d'autres tâches.

Dans la programmation synchrone, chaque action se fait l'une après l'autre. Le thread attend qu'une action est finie avant de passer à la suivante.

↳ ex: Le thread attend que le CPU effectue une lecture de fichier. En attendant il ne fait rien !

⇒ Perte de temps

-

```
function serveCustomer (customer, done) {
```

```
    customer.placeOrder (menu, [error, order] => {
```

```
        cook.prepareFood (order, [error, food] => {
```

```
            customer.eatAndPay (food, done)
```

```
        })
```

```
    })
```

```
}
```

↳ → fct de callback pour customer.placeOrder

↳ → fct de callback pour cook.prepareFood

↳ Seront effectuées lorsque customer.placeOrder et cook.prepareFood seront finis.

\Rightarrow et \Rightarrow fct de callback lambda.

utiliser error comme premier paramètre de la fonction callback est une convention Node.js

\Rightarrow Fonction callback de serveCustomer

Elle est passée comme paramètre à serveCustomer et est invoquée par customer.eatAndPay pour lancer le callback.

Avec cette écriture le code devient très complexe à lire (callbacks imbriquées). Il y a plusieurs manières d'écrire plus simplement le code:

→ Promesses

→ Async/Await

Promesses /
function serveCustomer (customer) {
 return customer.placeOrder (menu)
 .then (order => cook.prepareFood (order))
 .then (food => customer.eatAndPay (food))
}

Ajax / Thunk /
compt.serveCustomer = async (customer) => {
 let order = await customer.placeOrder (menu)
 let food = await cook.prepareFood (order)
 let tip = await customer.eatAndPay (food)
 return tip
}

Node API and EventEmitters

EventEmitter :

emitter.emit() } Méthodes permettant d'invoquer
emitter.on() } des fonctions de callback quand
des événements surviennent.

ex:

```
emitter.on('data', (msg) => {  
    console.log(msg)  
})
```

↳ On définit quelque chose à effectuer lorsque
l'événement 'data' survient

```
emitter.emit('data', 'Hello World!')
```

↳ Fait survenir l'événement 'data'
⇒ Ici : Affiche "Hello world!"

ex2 : → pas compris



* exemples, il y ena plus dans la documentation.

Streams:

↳ Manière de "stream" des données en respectant le principe asynchrone et l'event loop

3 types :

→ Readable

→ Writable

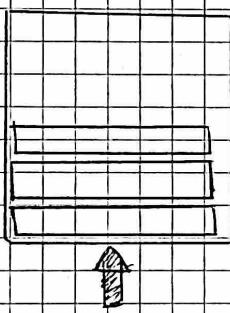
→ Read/Write (Duplex, Transform)

Readable Stream:

Événements : readable, data, end, error, ...

Méthodes : read, pause, resume, destroy, ...

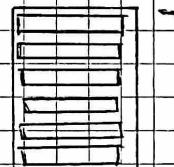
Propriétés : readable, readableLength, ... *



Il faut imaginer le stream comme un ensemble de données venant d'un fichier/réseau (✉) qui sera lu par notre code (💻).

Imagineons une boîte qui se remplit de données (✉) qui arrivent. La manière dont les données arrivent génère des événements. En réponse à ces événements on va effectuer des actions (ex: lire, pauser, ...)

ex: On ne lit pas assez vite, les données commencent à "saturer en tampon". En dépassant un certain seuil de données non lues le stream bloquera les données entrantes et nous en informera.



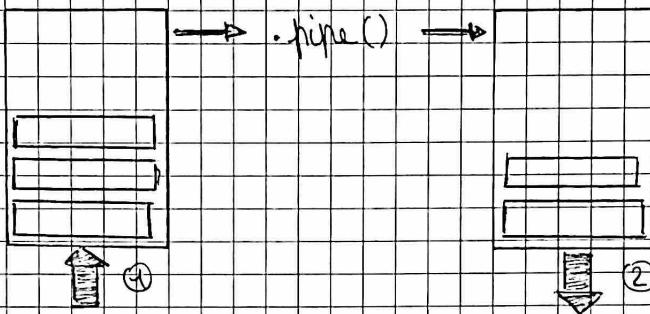
→ on bloque l'arrivée de données

Writable stream:

↳ Idem mais dans l'autre sens. A ses propres événements, méthodes et propriétés.

Piping streams:

↳ Relier un readable stream à un writable stream.



ex: lecture d'un fichier ①, Affiche sur une page web ②

ex2: Réception d'un Upload HTTP ①,

Ecriture sur disque ②

Exemples d'utilisation pour des API:

fs. ReadStream, fs. WriteStream

↳ Lecture/Ecriture dans des fichiers (file system)

http. ClientRequest, http. IncomingMessage,

http. ServerResponse

↳ Utilisé pour HTTP

zlib. createGzip()

↳ On écrit sur le stream, les données sont compressées et sont accessibles en lecture.

On peut pipe ensemble autant de stream qu'on veut, tant qu'on commence avec un Readable stream et qu'on termine avec un Writable stream.

Defining our application and Managing dependencies:

Modularizing your application:

Mieux vaut répartir notre code en plusieurs fichiers plutôt que de tout mettre dans un seul.

On doit dans un fichier définir quels éléments pourront être accessibles depuis un autre fichier avec `module.exports = ...`

↳ On peut partager des variables, fonctions, classes,...

Dans les fichiers voulant accéder à ces éléments on définira des constantes en utilisant `require(...)`

↳ On indique vouloir accéder aux éléments du fichier `...` qui sont partagés.

Convention: fonctions → camelCase) → vérifier
classe → PascalCase /

⚠ `...` → path relatif, ex: `./cook`

↳ va chercher le fichier `cook.js` dans ce projet.

Require permet aussi d'appeler du code tiers, des librairies. ex: `const moment = require('moment')`

↳ librairie moment

NPM and application dependencies:

NPM ⇒ Node Packaging Manager

↳ Composé de deux choses :

- 1) Un repository de package où on peut installer des packages.
- 2) Un CLI

| nppm init

↳ Initialise un projet Node.js.

Crée un fichier package.json de base. package.json est un fichier présent dans tout projet Node.js.

Il contient plein de métadonnées

| nppm install --save moment

↳ Installe le module 

Télécharge le module depuis le repository de package et le place dans node-directory.



--save => ajoute le package comme dépendance du projet dans package.json (dans la partie 'dependencies').

Ce module sera obligatoire pour faire tourner le projet.

--save-dev => Module sera utilisé dans le développement du projet (ex: testing, ...).



Il sera noté dans 'devDependencies' dans package.json.

| nppm install

↳ Télécharge tous les modules besoins pour le projet qui sont notés dans package.json

Souvent les modules ont d'autres dépendances eux-mêmes. Quand on télécharge les modules on télécharge aussi leurs dépendances.

Ex: 2 modules moment, et mocha

=> en vrai 180 modules, 164kb de code

Tous n'ont pas la même qualité (bug, problème de sécurité, ...)

* npm → permet de faire tourner plusieurs versions de Node

Autre solution à npm: yarn

Assembling a development toolset

Installing Node.js / testing applications:

Installation → Soit sur le site nodejs.org,
soit avec homebrew : brew install nodejs

*

Il y a aussi des images officielles dispo sur DockerHub.

Express → framework très utilisé pour créer des API



Tester Node.js:

- Mocha → peut être installé de manière globale ou comme une dépendance d'un projet

- Chai → librairie d'assert

- ▷ Il existe un module assert built-in
mais Chai plus facile, complet, ...

- Sinon → Spies, Stubs and Mocks

- Istanbul → Code coverage

- ▷ Permet d'indiquer le pourcentage de coverage + les lignes pas couvertes.

Debugging:

Bien que des console.log peuvent être bien pratique pour le debugging il ne faut pas ignorer le debug mode.