

Pluralsight: Node.js

Node.js: Getting started → Beaucoup de code
(disponible sur GitHub)

Introduction

What is node.js?

↳ Du Javascript en backend. Ce n'est pas Node.js qui fait tourner le JS. C'est une machine virtuelle comme V8 ou Chakra. Node.js n'est que le coordinateur.

↳ Node.js passe le code à une VM qui l'exécute.

Why node?

↳ Déjà vu dans mes recherches.

Node s'appelle comme ça car c'est son principe. On crée des petits codes (Nodes) qui on met ensemble pour créer de plus grosses applications.

Callback

↳ Fonction que Node appellera plus tard à un certain point dans le temps.

ex: function cb(data) {
 // do something
}
} → fonction callback

someAsyncMethod(cb);
est appellée quand someAsyncMethod est finie

ex: function Alexandre(coffee) {
 // drink coffee
}
}

=> On demande à Starbucks de faire un café.
Une fois qu'il est fini je le bois.
(appel de la fonction Alexandre)

Starbucks.makeMeACoffee(Alexandre);

Autre structure: Promesses:

↳ On demande quelque chose, on reçoit quelque chose d'autre et on nous promet que cet autre objet deviendra peut-être ce qu'on a demandé.

Il y a deux cas:

1) Ça devient ce qu'on a demandé (réussite) ①

2) Ça ne devient pas ce qu'on a demandé (échec) ②

Analogie de la poule:

↳ On demande un poussin et on reçoit un œuf.

const egg = chicken. makeChick();

egg. then (chick => raiseChick) ①

egg. catch (badEgg => throw badEgg) ②

D'autres analogies sur `Promise.all`/code-analogies.

What you get when you install Node.js

| ... \$ mode

↳ Lance une console où on peut taper du JS.

| ... \$ npx

} → On verra plus tard

| ... \$ npx

① Fichiers tests sur GitHub → Je les ai cloné localement

Getting started with Node

Node's REPL Mode:

↳ REPL \Rightarrow "Read Eval Print Loop"

\Rightarrow On accède au REPL en tapant 'node' dans le terminal.

Très efficace pour rapidement tester des commandes JS et Node.js simples en ligne de commande.

L'affichage se fait automatiquement \rightarrow pas de console.log()

▷ Certaines actions ne renvoient pas de valeur (undefined)

ex: let arriver = 42; \rightarrow ne renvoie pas de valeur.

①

ctrl + L \Rightarrow clear le REPL

REPL est intelligent et passe en mode multiligne

si on en a besoin (ex: définit° d'une fonction).

On peut aussi faire du multiligne en tapant '.editor'.

On rentre dans un éditeur où on tape autant de lignes

qu'on veut. \rightarrow ctrl + V \Rightarrow quitte l'éditeur et lance le code

②

.help \Rightarrow affiche toutes les commandes disponibles.

.done m \Rightarrow sauve les commandes précédemment

tapées dans le REPL dans un fichier m.

.load m \Rightarrow lance dans REPL le code du fichier m.

TAB and Underscore:

Dans REPL: • TAB \Rightarrow auto-complétion

• 2x TAB \Rightarrow affiche des propositions

\Rightarrow On peut voir toutes les possibilités pour les commandes JS, méthodes sur un objet ...

Underscore \Rightarrow stocke la dernière valeur générée. *

à une variable
on peut attribuer la valeur de -
38
37
36
35
34
33
32
31
30
29
28
27
26
25
24
23
22
21
20
19
18
17
16
15
14
13
12
11
10
9
8
7
6
5
4
3
2
1
0
Math.random()
const random =
*

Certains modules font vraiment partie de Node.js
Certains modules sont built-in. Ils peuvent être
directement utilisés dans le REPL mais dans un
projet classique on doit les require.

Executing scripts:

1... \$ node file-path

↳ Exécute le fichier file-path (path absolu ou relatif).

Si il y a une tâche continue (ex: loop) le fichier
continuera à tourner. Sinon Node exécutera
le script et quittera tout de suite après.

Créer un serveur:

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

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

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

```
});
```

① On require le module http

② On crée un serveur avec http.createServer()

Il prends un argument appelé ''Request listener''

↳ Fonction que Node exécute chaque fois qu'il y a
une requête effectuée sur le serveur web.

req => objet 'request' → requête effectuée au serveur

res => objet 'response'

Activer un serveur:

```
server.listen (4242, () => {
```

```
    console.log ('Server is running ...');
```

```
});
```

On fait écouter le serveur sur un port de notre machine.

Le deuxième paramètre donné ici est une fonction de callback qui s'exécute après que le serveur soit bien en train de tourner.

server.listen() garde le processus Node tourner.

C'est la tâche qui garde Node occupé et l'empêche de quitter.

① Après un changement → restart le serveur

↳ Il existe une autre manière => plus tard



ES6 → Dynamic script module management:

↳ Différent du "Common JS module management system"
(système de management de modules).

2 manières:

- 1) Configurer tout le projet => plus tard
- 2) Utiliser un fichier .mjs au lieu de .js



On utilise plus require mais import!

↳ `import http from 'http';`
↳ nom du module ou
nom path.

`import {createServer} from 'http';`

↳ named import : on import seulement certaines
choses et pas tout le module.

On utilise plus `http.createServer()` mais
`createServer()`.

plusieurs setTimeout
démarrant tous-en
même temps

Tous
*

Working with timers:

↳ setTimeout, setInterval

setTimeout (id, ms)

↳ attend \approx millisecondes puis effectue la fnct \approx

⚠ Le compteur démarre quand on lance le script.*

Si la fnct \approx a besoin d'arguments on les donne
comme arguments supplémentaires pour setTimeout.

↳ setTimeout (id, ms, ...)

setInterval (id, ms, ...)

↳ Attends \approx millisecondes puis effectue \approx toutes les
 \approx millisecondes indéfiniment.

clearTimeout (id)

↳ Remets à zéro un timer.

\approx → id d'un timer renvoyé par setTimeout

clearInterval (id) → Idem pour les intervalles

setImmediate (...) = setTimeout (... , 0)

clearImmediate (id) → idem pour les setImmediate

⚠ Un timer ne commence pas vraiment quand on
lance le script mais après qu'il ait fini toutes les
autres choses dans le script

↳ Imaginons une grosse boucle. Le timer sera
beaucoup retardé.

Node's command line interface:

... \$ node -v

↳ Affiche la version de Node.js

Il existe plein de commandes intéressantes qu'on peut voir avec : ... \$ node -h | less

... \$ node --help-options

↳ Affiche toutes les commandes d'options du Vd.

Il y a aussi plein de variables d'environnement.

On peut les voir à la fin de la commande d'aide.

The "process" object:

L'objet process permet d'accéder à des variables de l'OS déjà existantes ou qu'on a créée.

process.env.USER => utilisateur du pt de vue de l'OS
↓
variable

On peut définir des variables dans REPL:

ex: | export val1=100

On les utilise de la même manière : process.env.val1

On peut aussi définir des valeurs en même temps que de lancer un script.

↳ | \$ val1=10 val2=20 node my-file.js

autre manière de passer des arguments :

↳ via process.argv

I/O :

↳ stdout, stdin, stderr

process.stdout.write ("Hello World \n");

= console.log ("Hello World");

process.stdin.read();

↳ lit une valeur dans le stdin

On peut aussi pipe : process.stdin.pipe(process.stdout)

⇒ On verra les stream plus tard

process.exit()

↳ Quitte le processus Node qui tourne.

process.on (~~évenement~~, ~~fonction~~)

↳ Quand ~~évenement~~ se passe sur process, effectue ~~fonction~~.

ex: process.on ('exit', (...)) => { ... }

↳ quand le processus quitte

On peut utiliser "on" sur plein de choses

ex: process.stdin.on ('readable', ...)

↳ Quand on peut lire sur stdin ...

Modern Javascript

EcmaScript and TC39:

EcmaScript → ensemble de norme concernant les langages de type 'script'.

TC39 → Groupe maintenant et faisant évoluer JS.

Node.js => on peut seulement accéder aux features implementées dans JS pas celles en cours de création.

↳ Il ya moyen quand même (flags, harmony) ou compilation des codes des features via 'Babel'.

Babel => transforme du code JS "next-gen" en code JS compatible sur les browsers.

Variables and Block scopes:

{ } → Block scopes (if, else, ...) ≈ function scopes

Si on définit une variable avec var

- dans un function scope, on ne peut pas y avoir accès en dehors du scope.
- dans un block scope, on peut.

⇒ Différence

Définir avec let n'a pas cette différence. On ne peut pas y accéder en dehors du scope.

Const → idem

Const → pointe vers un objet, string, int, ...

On ne peut pas redéfinir une constante mais on peut changer l'objet ou l'array vers lequel la constante pointe.

A String → ne peut pas être partiellement modifié

Arrow functions:

↳ Autre façon de définir des fonctions.-

(\dots) \Rightarrow {
↓
param. ↓
ce que fait la fact}

ex. $(a, b) \Rightarrow \{ \text{return } (a+b) \}$
 $() \Rightarrow \{ \text{console.log ("Salut")} \}$
↳ pas de param.

On peut attribuer ces fonctions à une variable. Mieux
vaut utiliser const :

const squared = (a) => { return (a * a) }

Si un seul param. \Rightarrow on peut retirer les ()

Si une seule ligne de code => ... { }

Ex :

const square = (a) => { return a*a; }

pent derenir : $a \Rightarrow a^* a$

→ utile pour des méthodes utilisant des fonctions comme paramètres.

Valeur de "this":

fonction classique => "this" est celui qui appelle la fonction
fonction fléchée => "this" est celui retournant dans le
scope de la fonction.

On Denna "this" "plus land".

Objects literal:

↳ Définir une fonction, un objet, ... en l'attribuant à une variable sans utiliser un créateur (ex: New Object).

Exemple pour les objets

const mystery = "answer";

const PI = Math.PI;

const obj = {

p1: 10,

p2: 20,

f1(){}},

f2: () => [...],

[mystery]: 42, *

PI, **

};

* En mettant le nom d'une variable entre []

on crée un attribut de l'objet avec comme nom

la valeur de la variable.

↳ obj.answer → égal 42

** → Nom de l'attribut = nom de la constante de la valeur

↳ On peut l'écrire qu'une fois.

Destructuring and Rest / Spread:

Destructuring \rightarrow l'utilisation d'attributs d'objets sans utiliser la syntaxe : objet.attribut.

const { PI, E, SQRT2 } = Math;

\hookrightarrow équivalent à : const PI = Math.PI

const E = Math.E

const SQRT2 = Math.SQRT2

const circle = {
 label: 'circle1',
 radius: 2,
};

Indique que le paramètre n'est pas obligatoire
valeur par défaut ↑

const circleArea = ({ radius }, { precision = 2 } = {})
 $\Rightarrow (\text{PI} * \text{radius} * \text{radius}).toFixed(\text{precision})$;

console.log(circleArea(circle));

Rest:

3 points

const [first, second, ...restOfItems] = [10, 20, 30, 40, 50];

\hookrightarrow first = 10, second = 20, restOfItems = [30, 40, 50]

\Rightarrow Marche aussi avec les attributs d'objets

Spread:

\hookrightarrow Permet de copier les valeurs dans d'autres variables.

ex: const newArray = [...restOfItems]

\hookrightarrow newArray = [30, 40, 50]

3 points indique qu'on prends le reste des valeurs
et qu'on les met dans ...

Template strings:

↳ String délimités par des "back-ticks" =>

On peut écrire sur plusieurs lignes
et renouer le string dynamique.

▷ Ressemble à une apostrophe !

```
ex: const html = `

<div>
    ${Math.random()}
</div>
`
```



On introduit une variable avec \${ }

Classes:

↳ Template / blueprint pour définir des structures et comportements qu'on partage entre différents objets.

=> Ressemble aux classes Python

Promises and Async/Await:

```
function fetch (...){ ... }  
                _____  
                |_____ contient une instruction qui définit data.
```

Promesses:

```
fetch("https://curry.javascript.com/")
    .then(data => {
        console.log(data.length);
    });

```

Déux manières différentes d'écrire

Async/Await:

```
(async function reold () {
    const data = await fetch(...);
    console.log(data.length);
})()
```

=> on définit une fonction
async
En indiquant dans celle-ci qu'on effectue fetch() en asynchrone, on
réfère à tourner la fonction
async sur le thread quand fetch() sera fini

Packag = module

NPM: Node Package Manager

npmjs.com → site web npm où on peut retrouver plein de packages ⇒ c'est un registry.

NPM → Package Manager officiel de JS.

The NPM command:

| ... \$ npm install -g npm

↳ met à jour npm

node_modules → dossier contenant les modules installés

package.json

↳ Contient plein d'informations sur les packages.

Se remplit automatiquement lorsqu'on installe un paquet.

package.json / package-lock.json:

package.json:

↳ On doit rarement le changer à la main.

"name" → nom. Si on partage sur npm il doit être unique dans tout le registry.

"dependencies" → packages qu'un projet a besoin pour fonctionner.

⚠ On ne partage pas le dossier "node_modules".

On partage "package.json".

et sub-dependencies

Pour installer tous les dépendances d'un projet:

| ... \$ npm install

⚠ Installe les versions mises dans package.json

| ... \$ npm install -e

↳ Installe un package et le met dans le dependencies

| ... \$ npm install -D

↳ package sera mis dans les dev-dependencies.

① Vu commandes un peu différentes dans précédent cours.

| ... \$ npm help install

↳ Affiche l'aide pour npm install

② ... \$ npm install --production

↳ Installe les packages de dépendances mais pas ceux de dev-dependencies.

| ... \$ npm init

↳ Va lire - créer un package.json

On réponds à certaines questions. NPM prends les valeurs par défaut si on utilise le flag "-yes".

③ Semantic Versioning (SemVer):

↳ Manière dont on écrit des numéros de versions.

Un peu compliqué.

remover.npmjs.com => Calculateur / traducteur de SemVer

~ 2.16.3 => 2.16.X , X ≥ 3

^ 2.16.3 => 2.Y.X , Y,X ≥ 16.3

Installing and using NPM packages:

↳ Moyen d'installer des packages en local ou global.

| ... \$ npm install -g ↳

↳ Installation globale de ↳. On pourra l'utiliser partout sur notre machine.

Creating / Pushing NPM package:

Quand on require un folder, Node.js cherchera un fichier index.js dans ce folder.

| ... \$ npm login

↳ Se connecter à un compte de npmjs.com

| ... \$ npm publish

↳ Partager un paquet sur npm.

On se trouve dans le folder (dossier) du paquet qu'on veut partager quand on fait la commande.

NPX and the NPM run scripts:

On peut définir dans le package.json des scripts.

```
"scripts": {  
  "start": "node server.js"  
  "test": "jest"  
  "check": "eslint server.js"  
}
```

On lance les scripts : ... \$ npm run ↳

Les commandes ↳ seront alors effectuées.

ex: ...\$ npm run start ↳ ...\$ node server.js

⚠ Certains noms de scripts sont spéciaux. On ne doit pas utiliser "num" lorsqu'on veut lancer le script.

ex: ... \$ npm start ≡ ... \$ npm run start

... \$ npm test ≡ ... \$ npm run test

jest:

↳ Framework de test Javascript. Fonctionne avec React.

Simple!

ESLint:

↳ Analyse le code pour trouver des problèmes.



Airbnb => style guide pour ESLint. Fonctionne avec React.

On ne peut pas lancer les commandes "jest" et "eslint" de base sans passer par les scripts NPM. C'est parce qu'ils sont installés de manière locale.

↳ Solution : utiliser NPX => ... \$ npx jest

⚠ Il faut configurer ESLint => ... \$ npx eslint --init



| ... \$ npm help npm-scripts

↳ Pour voir les noms de scripts spéciaux

Updating NPM packages:

| ... \$ npm ls

↳ Affiche les dépendances installées et les sous-dépendances.

| ... \$ npm show ~~##~~ versions

↳ Affiche toutes les versions pour le module ~~##~~

| ... \$ npm outdated

↳ Affiche les versions qui ont à être installées avec "npm update"

1... \$ npm update

↳ Mettre à jour les packages selon les contraintes
dans package.json (`^...`, `<...`, `=...`)

Autre package manager => Yarn (yarnpkg.org)

Modules and Concurrency

Defining and using Node modules:

▷ Dans JS, le mot "arguments" représente l'ensemble
l'ensemble des arguments envoyés dans une fct

↳ ex: function my_funct () {
 console.log;
}

my_funct (3, 9, 7, 2); ↴

affiche les arguments de la fct

⇒ [Arguments] { '0': 3, '1': 9, '2': 7,
'3': 2 }

Node enveloppe les fichiers d'une fonction.

Si on fait un console.log (arguments) dans un fichier.

- browser => erreur
- Node.js => 5 arguments

Un fichier Node a 5 arguments : exports, module,
require, --filename, --dirname.

require => permet de 'require' d'autres modules

--filename => path du fichier

--dirname => path du directory du fichier.

La fonction qui enveloppe le fichier renvoie une valeur :
module.exports (caché, n'est pas écrit dans le code).

↳ 'API' du module

⚠ "exports" est un alias de "module.exports".

Pour récupérer l'API d'un module (les données qu'il renvoie). Dans un autre fichier on effectue un 'require' :

const moduleAPI = require ('_____')

↳ Pathname

(absolu ou relatif)

O

On définit des données dans module.exports en définissant des variables de exports.

ex: exports.a = 47;

module.exports.b = 22;

⇒ module.exports = {a: 47, b: 22}

C

On peut transformer module.exports (de base un objet) en fonction.

↳ module.exports = (...) = {...}

⚠ exports = (...) = {...}

Exemples of modules API:

L'API d'un module peut-être un objet (de base), un array, un string ou une fonction.

Il suffit de redéfinir son type en réattribuant module.exports (⚠ pas exports tout seul).

⚠ Pour un string dynamique on doit utiliser une fonction.

Node's global object :

Vu que les fichiers sont enveloppés par une fonction,
les variables qu'on définit globalement dans un fichier
ne le seront en fait pas (\Rightarrow function scope).

\hookrightarrow Solution : objet global \Rightarrow objet partagé par tout le monde.

En définissant une valeur de 'global' tout le monde
peut y accéder.

ex: `global.answer = 42;` \rightarrow dans fichier 1

`console.log(answer);` \rightarrow dans fichier 2



Δ Ne PAS faire ça !

The event loop :

\hookrightarrow Ce que Node utilise pour gérer des actions asynchrones
pour qu'on ne doive pas gérer des threads.

C'est une boucle infinie qui tourne jusqu'à ce qu'un
événement se produise, une interruption arrête
l'event loop et que l'action liée se produise.



ex: `setInterval(() => {`

`console.log('Salut');`

`}, 5000);`

\Rightarrow Event loop tourne à l'infini et tous les 5 sec.

on a une interruption qui fait un `console.log`.

Errors VS Exceptions:

Error => application is not prepared to deal.

Excepto => " " is prepared to deal

Gérer des erreurs, 2 manières:

Try / Catch:

```
try {  
    //  
}  
catch (err) {  
    //  
}
```

=> On place le code problématique dans le try → //

Si une erreur se produit le code ne s'arrête pas et le code // est effectué.

Problème: on ne peut pas cibler le type d'erreur.

Le code // s'effectuera, peu importe le type d'erreur.

Try / Catch + condition sur l'erreur:

```
try {  
    //  
}  
catch (err) {  
    if (err.code == //) {  
        //  
    }  
    else {  
        throw err;  
    }  
}
```

→ code de l'erreur

→ Si on a pris prévu une exception renvoie une erreur.

=> Fait une vérification sur le code d'erreur (err.code) si il y en a une. En fonction effectue un certain code.

Node clusters :

↳ On peut faire tourner plusieurs mœurs ensemble sur un ou différents serveurs. Pour les gérer on utilisera un Master.

⇒ Voir cours avancé

Node's asynchronous Patterns :

Callbacks :

↳ Dernier paramètre d'une fonction asynchrone représente la fonction de callback. C'est la fonction qui sera effectuée lorsque la fonction asynchrone sera terminée.

ex: const fs = require('fs');

```
fs.readFile(__filename, function (err, data) {  
    console.log('File data is', data);  
});  
console.log('TEST')!
```

err, data

↳ err => objet erreur si une erreur se produit
Si pas d'erreur → null

↳ data => données générées par la fonction asynchrone

Problème : callbacks imbriqués peuvent être difficile à lire.

⇒ "Pyramid of doom"

Promesses :

↳ On fait une promesse qui une fonction finira.

En attendant on libère le thread à d'autres instructions.

On utilise soit Promiseify, soit les promesses d'un module.

On définit une fonction asynchrone (async) et à l'intérieur on utilise des await.

A chaque await, la fonction s'effectue en dehors du thread principal. Dès que la fonction est terminée la func renvoie sur le thread principal et on continue les instructions (jusqu'à la fin ou au prochain await).

ex promisify:

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

const readfile = util.promisify(fs.readFile); ①

async function main () { ②
    const data = await readfile ('filename'); ③
    console.log ('file data is ', data);
}

main();
```

○

de l'ex de :

① On fait en sorte que fs.readFile puisse fonctionner avec des promesses.

② Fonction asynchrone main()

③ Fonction await

ex. Promesses d'un module:

```
{ const {readFile} = require ('fs').promises;
```

↳ A écriture équivalente :

```
const readFile = require ('fs').readFile.promises;
```

↳ vérifier !!!

Event emitters:

↳ On peut créer des événements et en fonction des événements on programme des actions.

Initialiser EventEmitter:

```
const EventEmitter = require('events'); built-in  
const myEmitter = new EventEmitter(); ②
```

① Requiert le module events

② On crée un nouvel objet émetteur d'événements.

Définir un événement:

```
myEmitter.on(name, callback);
```

↳ Sur l'événement défini par la string ~~#~~ on invoque la fn de callback ~~#~~.

Faire survenir un événement:

```
myEmitter.emit(#);
```

↳ On fait survenir l'événement défini par la string ~~#~~. Effectue la fn de callback définie pour myEmitter.on().

! Si on a pas défini l'événement rien ne se passe.

⇒ Les événements sont représentés par un string.

C'est comme leur nom, leur identifiant.

Working with Web Servers

Hello World... The Node's version:

Les fonctions de type listener ont deux arguments: req, res

On peut les nommer comme on veut mais c'est une convention. req et res sont des objets!

req => représente la requête

res => représente la réponse

Fonction listener => sera invoquée chaque fois qu'il y aura une requête.

On a déjà vu comment créer un serveur et attribuer un callback chaque fois qu'il y a une requête.

compt server = http.createServer(callback);

=

compt server = http.createServer();

server.on('request', callback);

équivalent

serveur est en fait un event emitter. Les deux versions de code sont équivalentes. On définit la fonction de callback quand l'événement 'request' survient.

server.listen(port, callback)

L'écoute sur un port port. callback est une fn de callback.

=> déjà vu!

Monitoring files for changes.

Pour mettre en place les modifications qu'on a fait dans un fichier il faut redémarrer le serveur.

Il ya plusieurs possibilités pour faire en sorte que le serveur se relance automatiquement. Comme par exemple modemon.

La commande 'modemon' enveloppe la commande 'mode'.

En utilisant 'modemon' pour lancer un serveur au lieu de 'mode', chaque fois qu'on modifie un fichier et qu'on le sauve le serveur se relance.

↳ ex: !... \$ modemon index.js

Intéressant d'en installer en global

↳ !... \$ mode i -g modemon

The req and res objects:

Une bonne manière pour voir de quoi est composé req et res et de les afficher avec console.log lors d'une requête.

En utilisant : console.dir({req, {depth: 0}})
on affiche seulement le premier niveau.

⚠ req est de type 'IncomingMessage' de http.

⇒ Ce qu'on doit chercher pour la documentation.

res est de type 'ServerResponse' de http.

req et res sont des streams!

Node web frameworks:

Express.js

On doit d'abord installer Express : `npm i express`

`const express = require('express');` ①

`const server = express();` ②

`server.listen(4242, function);` ③

↳ callback

① Le require renvoie une fonction qu'on utilisera pour créer un serveur (2).

② Le listen est très semblable à celui de http.

La différence avec http c'est qu'on ne définit pas un seul request listener mais plusieurs. Un pour chaque url

```
server.get('/', (req, res) => {  
  res.send('Hello World!')  
});
```

~~'/'~~ ⇒ url

~~res.send('Hello World!')~~ ⇒ code à effectuer lorsqu'on rejoins l'url ~~'/'~~ sur le serveur

`res.send()` => équivalent à `res.write()` de http. *

On ne doit pas mettre de `res.end()`, express le fait automatiquement.

```
ex: server.get('/', (req, res) => {  
  res.send('Hello Express!');  
});
```

```
server.get('/about', (req, res) => {  
  res.send('About...');  
});
```

* Contidument à `res.write()` on peut envoyer des objets (ex: objets JSON)

Autres frameworks => KOA, rails (inspiré de rails), Meteor, ...

Using template languages:

↳ PUG, handlebars, EJS, React avec JSX, ...

EJS:

On indique qui on utilise EJS:

```
server.set('view-engine', 'ejs');
```

Au lieu d'utiliser res.send() on utilise:

```
res.render(...);
```

... → indique quel template (fichier .ejs) on envoie

Tous les fichiers .ejs se trouvent dans un dossier 'views'.

React:

↳ On peut faire du templating en frontend et backend.

Working with operating systems

↳ J'ai pris moins de notes. J'en aurai plus besoin pour le projet.

The os module:

The fs module:

The child-process module:

↳ Pour lancer des commandes shell.

Debugging node applications:

```
... $ node --inspect-brk app.js
```

↳ Lance le débogage pour le fichier app.js. *

Collant sur l'URL
aura dans "Remote Target" le fichier dont on a lancé le débogage.
Collant sur "inspect" on a accès aux débuggeurs "chrome DevTools".
Collant sur "inspect" on a accès aux débuggeurs "chrome DevTools".