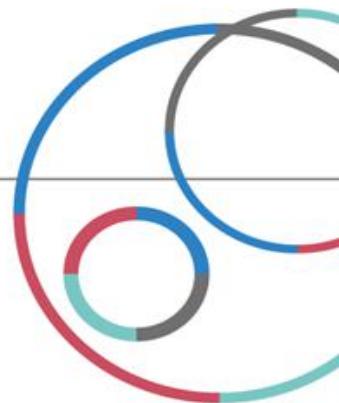




Notre expertise est votre avenir



Node.js



JVS-NOD

Sommaire

I.PRÉSENTATION	P. 3
II.INTRODUCTION	P. 7
III.GESTIONNAIRE DE PAQUETS NPM	P. 24
IV.CONCEPTS	P. 40
V.FLUX	P. 61
VI.MANIPULATION DES FICHIERS	P. 75
VII.PROMESSES	P. 86
VIII.APPLICATION EN LIGNE DE COMMANDE	P. 97
IX.APPLICATION WEB	P. 107
X.BASE DE DONNÉES	P. 132
XI.SOCKET.IO	P. 144
XII.TESTS	P. 160
XIII.OUTILS DE DÉVELOPPEMENT	P. 172
XIV.DÉBOGAGE	P. 179
XV.MISE EN PRODUCTION	P. 186

Introduction



- 1. Nom**
- 2. Société**
- 3. Titre / fonction**
- 4. Position**
- 5. Expérience en bases de données**
- 6. Connaissance du produit**
- 7. Attentes concernant le cours**



Objectifs



- 1. Penser et développer en asynchrone dans un environnement multi-utilisateurs.**
- 2. Maîtriser les API fondamentales fournies par Node.js.**
- 3. Approfondir NPM et la modularité.**
- 4. Accéder aux données depuis Node.js.**
- 5. Utiliser les modules Express et Socket.IO**
- 6. Déployer une application Node.js.**

© m2iformation

JVS-NOD

4

Participants et prérequis



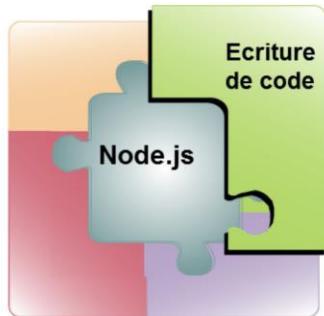
1. Prérequis :

- a. Avoir une connaissance pratique de JavaScript et jQuery ou avoir suivi le cours JVS-IN "JavaScript".

2. Public concerné :

- a. Développeurs, architectes, chefs de projets techniques.

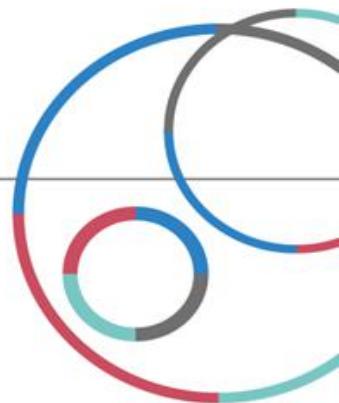
Plan de cours



- 1. INTRODUCTION**
- 2. GESTIONNAIRE DE PAQUETS NPM**
- 3. CONCEPTS**
- 4. FLUX**
- 5. MANIPULATION DES FICHIERS**
- 6. PROMESSES**
- 7. APPLICATION EN LIGNE DE COMMANDE**
- 8. APPLICATION WEB**
- 9. BASE DE DONNÉES**
- 10. SOCKET.IO**
- 11. TESTS**
- 12. Outils de développement**
- 13. DÉBOGAGE**
- 14. MISE EN PRODUCTION**



Notre expertise est votre avenir



Introduction



JVS-NOD



Définition de Node

1. Plateforme de développement libre et événementielle en Javascript orientée vers les applications réseau qui doivent pouvoir monter en charge.
2. Un interpréteur Javascript associé à un ensemble de bibliothèques permettant de réaliser des actions.
3. Son but : fournir un moyen **simple** de produire des applications **performantes** et **extensibles**.



Historique

1. Ryan Lienhart Dahl
2. Le moteur Javascript V8
 - Développé par Google
 - Moteur d'exécution de Node
 - Compile le Javascript en langage machine
3. 2009 : première version de Node (Dahl)
 - Uniquement sous Linux
4. 2012 : Dahl → Isaac Schlueter qui est à l'origine de npm

Success-stories

1. PayPal
 - Au départ deux équipes (une pour le code côté client, et l'autre pour le code côté serveur (en Java)).
 - Parallèlement, une troisième équipe commence à écrire en Node.js.

The diagram illustrates the PayPal success story. It shows two parallel paths leading to the same outcome. On the left, a blue box labeled 'Equipe Java 5 ingénieurs expérimentés' has a downward arrow pointing to a central box labeled 'Livraison au même moment'. On the right, a blue box labeled 'Equipe Node 2 ingénieurs novices' also has a downward arrow pointing to the same central box. Below this central box is another blue box labeled 'Fin du projet'. To the right of the central boxes is a pink box containing a list of Node.js benefits.

Node JS :

 - Développement deux fois plus rapide
 - Un tiers de lignes de code en moins
 - Presque moitié moins de fichiers
 - Deux fois plus de requêtes traitées par seconde
 - Réduction d'un tiers du temps de réponse moyen pour les requêtes
2. Groupon, Walmart, Xen Orchestra

© m2ifformation JVS-NOD



Programmation orientée composant

1. Consiste à utiliser une approche modulaire de l'architecture d'un projet informatique, ce qui permet d'assurer au logiciel une meilleure lisibilité et une meilleure maintenance.
2. Actuellement, plus de 100 000 paquets (voir chapitre sur le gestionnaire de paquets npm).

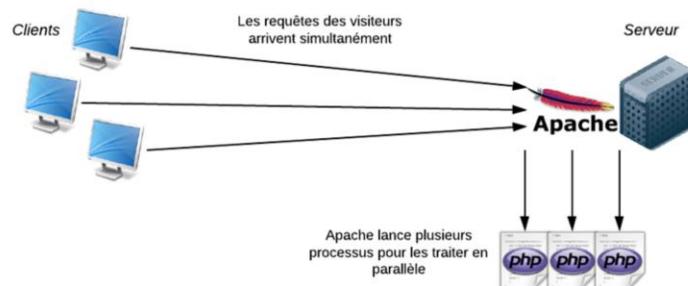


Node : les points forts

1. Le JavaScript peut être utilisé comme langage unique pour le serveur et pour la partie cliente, ce qui homogénéise le programme et facilite le développement.
2. Node permet de faire des requêtes asynchrones, ce qui permet une gestion des entrées / sorties de manière non bloquante, très pratique pour les applications qui ont besoin de « temps réel ».
3. Node, utilisé en tant que serveur Web, reste très performant (bien plus qu'Apache) en temps de réponse / requête simultanée.
4. Node intègre un système de dépôt de paquets, de nombreux plugins super-sympas sont proposés par une importante communauté.
5. Lesdits plugins sont facilement installables grâce à un gestionnaire de paquets conçu spécialement pour Node : NPM (Node Package Manager).
6. Avec Node, il est possible de tout configurer, qu'il s'agisse du port sur lequel vous répondez, des en-têtes que vous renvoyez... vous POUVEZ tout gérer...

Node.js vs Apache - PHP

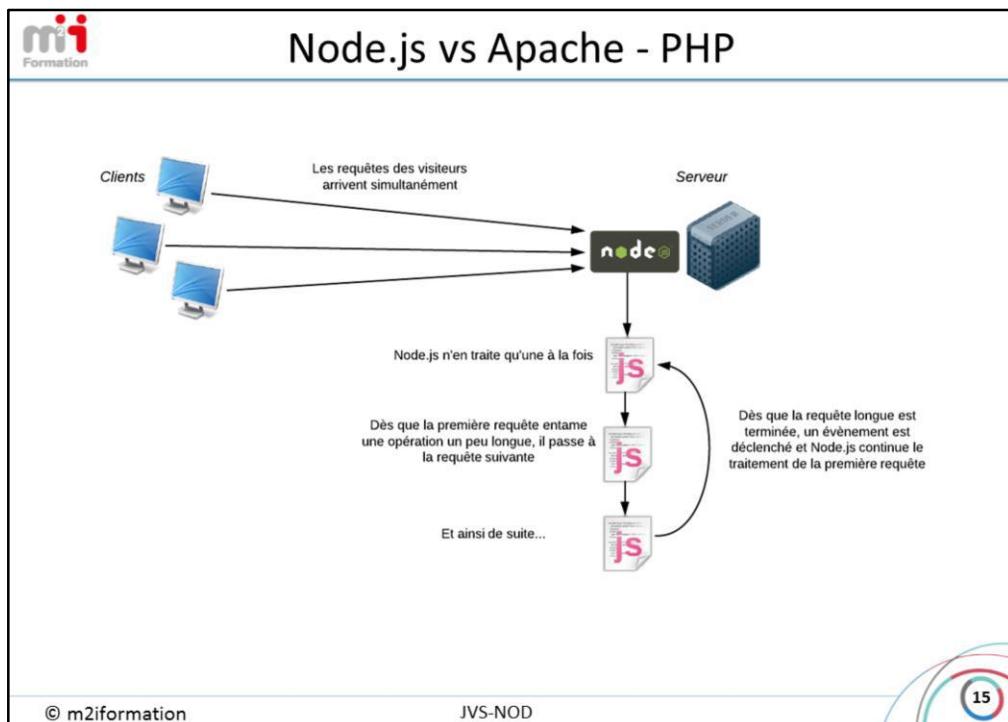
- Comme plusieurs visiteurs peuvent demander une page en même temps au serveur, Apache se charge de les répartir et de les exécuter en parallèle dans des *threads* différents.
- Chaque thread utilise un processeur différent sur le serveur (ou un noyau de processeur) :

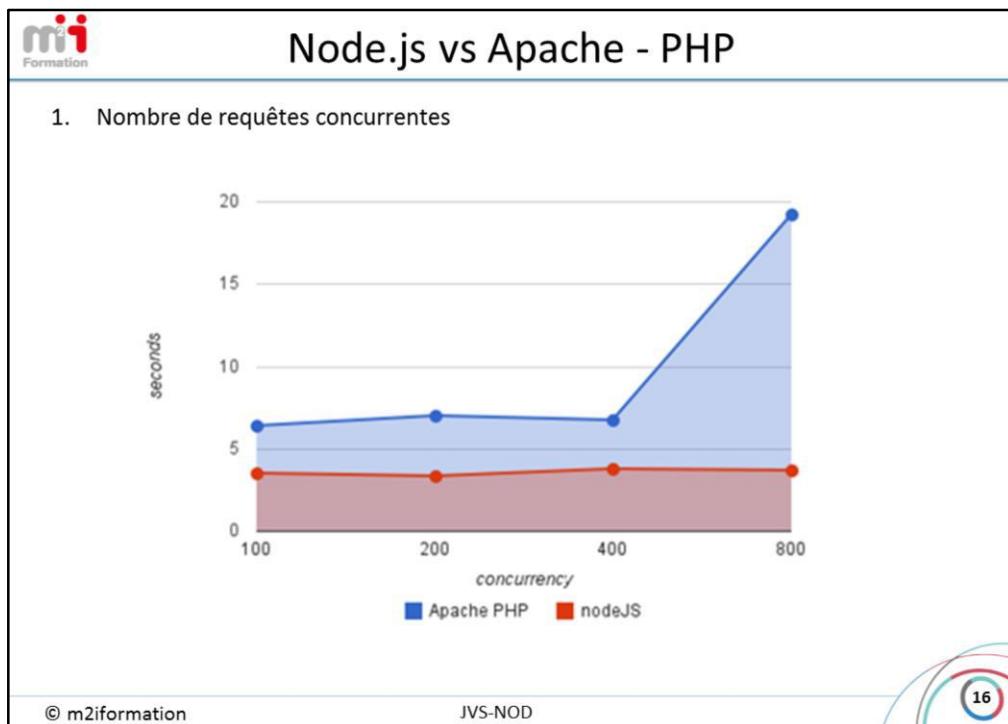




Node.js vs Apache - PHP

- Node.js est monothread, contrairement à Apache. Cela veut dire qu'il n'y a qu'un seul processus, qu'une seule version du programme qui peut tourner à la fois en mémoire.
- En effet, il ne peut faire qu'une chose à la fois et ne tourne donc que sur un noyau de processeur.
- Mais il fait ça de façon ultra efficace, et malgré ça, il est quand même beaucoup plus rapide !
- Cela est dû à la nature "orientée évènements" de Node. Les applications utilisant Node ne restent jamais les bras croisés sans rien faire. Dès qu'il y a une action un peu longue, le programme redonne la main à Node qui va effectuer d'autres actions en attendant qu'un évènement survienne pour dire que l'opération est terminée.

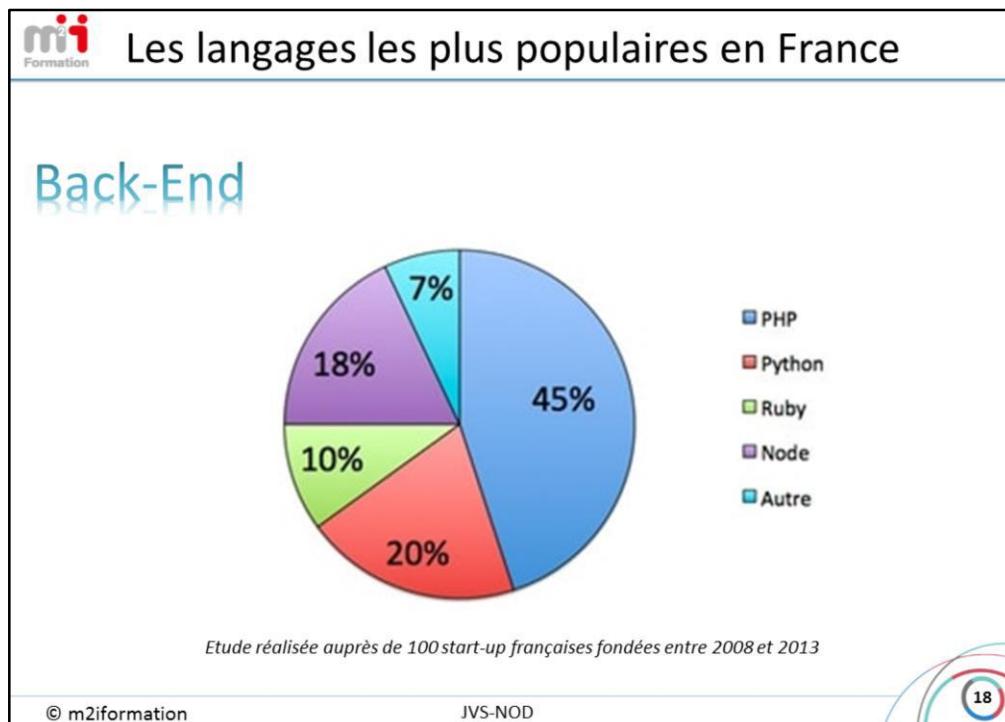


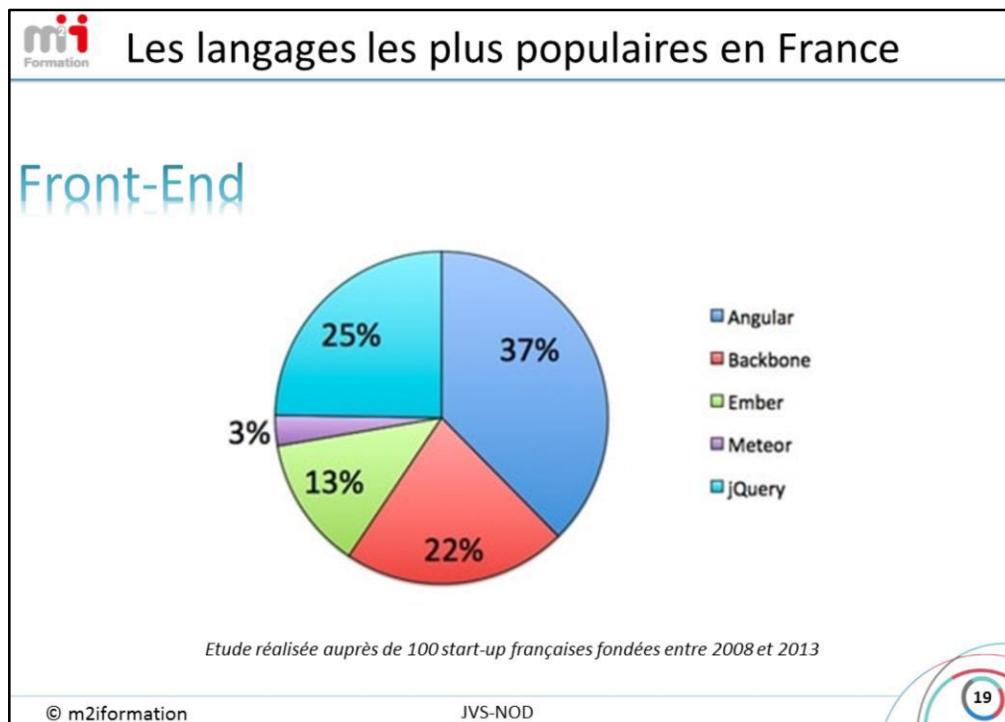




Node : les points faibles

1. Avec Node vous **DEVEZ** tout gérer, Node est une base, vous ne pourrez pas l'utiliser directement en tant que serveur Web / ftp / Webservices / autre... sans avoir à écrire du code.
Fort heureusement, les plugins simplifient beaucoup le travail (le paquet "**Express**" permet de transformer facilement Node en serveur Web).
2. Il existe encore très peu de docs francophones sur le sujet.
3. Pas mal de problèmes d'incompatibilités de versions entre Node et les paquets, la partie mise à jour est moyennement supportée.
4. La maturité des librairies à disposition côté serveur serait moins évoluée que celle de PHP, Python... (ce qui se tient, étant donné la jeunesse du produit).
5. En réalité, le plus gros point faible de Node, est précisément sa jeunesse : il est encore impossible de déterminer si le serveur est fiable sur le long terme en production.





INTRODUCTION

Installation de Node

1. Sur Windows : <https://nodejs.org/en/>

Node.js® is a JavaScript runtime built on Chrome's V8 JavaScript engine. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient. Node.js' package ecosystem, npm, is the largest ecosystem of open source libraries in the world.

Important security upgrades for recent OpenSSL vulnerabilities

Download for Windows (x64)

v4.4.5 LTS
Recommended For Most Users

v6.2.2 Current
Latest Features

Other Downloads | Changelog | API Docs Other Downloads | Changelog | API Docs

Or have a look at the [LTS schedule](#).

© m2iformation JVS-NOD



INTRODUCTION

Installation de Node

1. Sur Windows : <https://nodejs.org/en/>

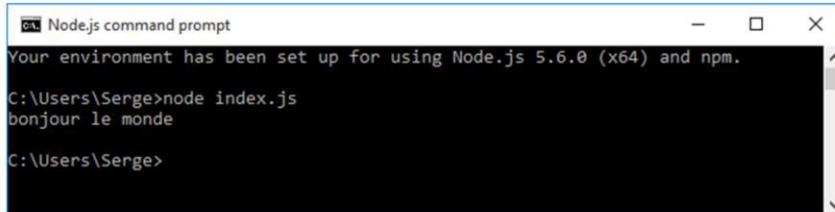
The screenshot shows the 'Node.js Setup' window with the title 'Custom Setup'. It displays a tree view of installation features: 'Node.js runtime' (selected), 'npm package manager', 'Online documentation shortcuts', and 'Add to PATH'. To the right of the tree, a detailed description of the selected feature ('Node.js runtime') is provided, including its size requirement (13MB) and subfeatures. At the bottom, there are 'Reset', 'Disk Usage', 'Back', 'Next', and 'Cancel' buttons. The 'Next' button is highlighted in blue.

© m2formation JVS-NOD

21

Test

1. Lancez la fenêtre de commande Node.js
2. Ecrire un script index.js :
 - `console.log("Bonjour le monde");`
3. Dans la fenêtre de commande Node.js :



```
Node.js command prompt
Your environment has been set up for using Node.js 5.6.0 (x64) and npm.

C:\Users\Serge>node index.js
bonjour le monde

C:\Users\Serge>
```

© m2ifformation JVS-NOD

22

Atelier : introduction



A cartoon illustration of a person with dark hair, wearing a purple shirt, sitting at a light blue desk. They are facing a vintage-style computer setup consisting of a CRT monitor on top of a keyboard unit. A mouse is connected by a cord between them. To the left of the computer is an open book with white pages and a dark cover. The person's hands are on the keyboard. The background is plain white.

© m2iformation JVS-NOD 23



Notre expertise est votre avenir



Gestionnaire de paquets npm



JVS-NOD

Introduction

1. Un paquet est un dossier contenant des ressources décrites par un fichier package.json
2. Exemple d'installation : npm install mon-paquet
3. Registre npmjs.org : plus de 60 000 paquets
4. Paquet global : contiennent des exécutables et sont généralement installés globalement : npm install --global mon-paquet



browserify
browser-side require() the node way
13.0.0 published a month ago by feross



express
Fast, unopinionated, minimalist web framework
4.13.4 published a month ago by dougwilson



pm2
Production process manager for Node.js appli...
1.0.0 published 2 months ago by tknew



grunt-cli
The grunt command line interface.
0.1.13 published 2 years ago by tkellen



npm
a package manager for JavaScript
3.7.1 published 3 weeks ago by lama

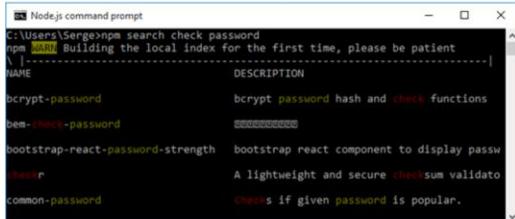


karma
Spectacular Test Runner for JavaScript.
0.13.19 published 2 months ago by dignifiedquire

© m2iformation JVS-NOD



Recherche du bon paquet

1. npm search <terme>
2. Exemple : npm search check password hash
3. Recherche sur npmjs.org
4. Alternatives :
 - <https://nodejsmodules.org/>
 - <http://www.nodetoolbox.com>

© m2iformation JVS-NOD



26



Versionnage

1. Un paquet peut exister sous différentes versions.
2. Une mauvaise gestion des dépendances peut avoir un impact négatif sur un projet.
Pour éviter cela, il est important de contrôler la version des dépendances utilisées.
3. npm est doté d'une fonctionnalité permettant d'utiliser la version la plus récente d'un paquet tout en assurant la compatibilité.



Gestion des dépendances

1. Dès qu'un paquet est utilisé dans un projet, il devient une dépendance. Il est alors nécessaire de la déclarer afin de permettre au programme d'être installé correctement.
2. Pour enregistrer les dépendances, il est nécessaire de convertir le projet en paquet, c'est-à-dire de créer un fichier package.json.
3. Cette étape peut être automatisée par la commande npm init.

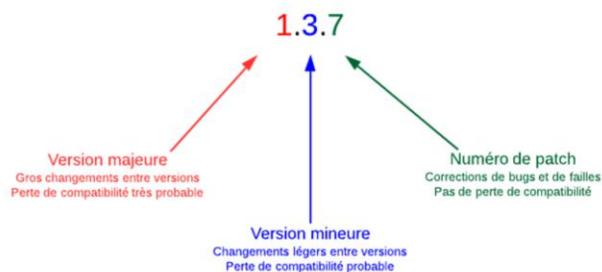


Gestion des dépendances

```
{  
  "name": "formation",  
  "version": "1.0.0",  
  "description": "Formation Node JS",  
  "main": "index.js",  
  "dependencies": {  
    "mon-module": "^2.0.1"  
  },  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "author": "Serge L'HOSTIS",  
  "license": "ISC"  
}
```

Le fonctionnement des numéros

- Pour bien gérer les dépendances et savoir mettre à jour le numéro de version de son application, il faut savoir comment fonctionnent les numéros de version avec Node.js. Il y a pour chaque application :
 - Un **numéro de version majeure**. Ce numéro change très rarement, uniquement quand l'application a subi des changements très profonds.
 - Un **numéro de version mineure**. Ce numéro est changé à chaque fois que l'application est un peu modifiée.
 - Un **numéro de patch**. Ce numéro est changé à chaque petite correction de bug ou de faille. Les fonctionnalités de l'application restent les mêmes entre les patchs, il s'agit surtout d'optimisations et de corrections indispensables.



Numérotation de version

1. Opérateurs de comparaison

- Les opérateurs `< <= > >=` peuvent être combinés avec `||` ou un espace pour former une expression définissant un ensemble de versions.

(ex : `1.2.7 || >=1.2.9 <2.0.0` correspond aux versions `1.2.7`, `1.2.9` et `1.4.6` mais pas `1.2.8` ni `2.0.0`)

2. Jokers

- Les métacaractères `*` – `~` `^` sont des raccourcis pour éviter des expressions complexes.

- Prendra toujours la dernière version disponible. Il peut être utilisé sur la version complète ou de manière partielle.

(ex : `*` équivaut à `>=0.0.0` c'est-à-dire la dernière version disponible, `1.*` équivaut à la dernière version de la branche 1 soit `>=1.0.0 <2.0.0`).

– détermine un intervalle inclusif

(ex : `1.2.3 – 2.3.4` équivaut à `>=1.2.3 <=2.3.4`).

`~` inclut la version mineure la plus récente mais sans changer de branche

(ex : `~1.2.3` équivaut à `>=1.2.3 <1.3.0`).

`^` inclut la version majeure la plus récente

(ex : `^1.2.3` équivaut à `>=1.2.3 <2.0.0`).



Ajout : dépendance de production

1. Il existe différents types de dépendance pour correspondre à chaque besoin.
2. Dépendance de production, nécessaire au fonctionnement du paquet :
 - npm install --save mon-module
 - Le fichier package.json contient maintenant la nouvelle dépendance avec sa version actuelle.

```
"dependencies": {  
    "mon-module": "^1.2.3"  
}
```

- Pour installer une version particulière :
➤ npm install --save mon-module@0.9.5



Ajout : dépendance optionnelle

- Le deuxième type correspond aux dépendances dites « optionnelles ». Ce sont des dépendances (assez rarement utilisées) qui peuvent étendre ou améliorer le fonctionnement du paquet mais qui ne sont pas nécessaires.
- npm install --save-optional mon-module

```
"optionalDependencies": {  
    "mon-module": "^1.2.3"  
}
```

- En cas d'erreur de récupération du paquet, npm continuera le processus d'installation puisque ce paquet est non essentiel au fonctionnement du projet.



Ajout : développement

1. Le dernier type concerne les dépendances de développement, utilisées pour la phase de développement du projet (tests unitaires ou fonctionnels).
 - npm install --save-dev mon-module

```
"devDependencies": {  
    "mon-module": "^1.2.3"  
}
```

- Ce type de paquet n'est pas récupéré par npm lors de l'installation en production du projet.

Mise à jour

1. Pour mettre à jour les dépendances tout en respectant les contraintes de versions présentes dans le fichier package.json : npm update
 - npm update mon-paquet,
 - A noter que npm update ne met pas à jour les contraintes de versions, elle ne fait que récupérer la dernière version du paquet,
 - Pour mettre à jour les contraintes, il est possible d'utiliser un outil comme npm-check-updates.

```
// Installation de npm-check-update
npm install --global npm-check-updates
// Mise à jour des paquets
ncu -u
```

© m2ifformation

JVS-NOD



35

Suppression

Il est utile de pouvoir supprimer facilement un paquet :

1. Pour une dépendance de production
 - `npm uninstall --save mon-module`
2. Pour une dépendance optionnelle
 - `npm uninstall --save-optional mon-module`
3. Pour une dépendance de développement
 - `npm uninstall --save-dev mon-module`



Listage des dépendances

Il est parfois utile de connaître la liste des paquets installés dans un projet :

1. npm ls
2. npm ls --depth 1 *permet de limiter l'affichage en fonction de la profondeur des dépendances*
3. npm ls --long *affiche des détails sur les paquets, comme leur description, adresse de dépôt...*



Installation des dépendances

1. Avec la commande npm install.
2. En ajoutant --production, les dépendances de développement ne sont pas installées.

m²i
Formation

Atelier : gestionnaire de paquets npm



© m2ifformation JVS-NOD 39



Notre expertise est votre avenir



Concepts



JVS-NOD



Modules

1. Unité de base de l'organisation du code dans Node.
2. Tout fichier JavaScript constitue un module.
3. Dans chaque module, il y a un objet global *module* qui le décrit. Cet objet possède plusieurs propriétés :
 - **id** : identifiant du module (souvent la même valeur que *filename*).
 - **filename** : chemin absolue du module.
 - **loaded** : booléen indiquant si le module est complètement chargé.
 - **parent** : le module qui a requis le module en question.
 - **children** : la liste des modules requis par le module en question.
 - **exports** : valeur qui est renvoyée lorsque le module est requis. Par défaut, cette propriété est initialisée à un objet vide {} qui peut être utilisé pour exporter plusieurs variables.



Modules de base

1. Node contient un certain nombre de modules intégrés qui forment l'API de la plateforme. Il en existe plus d'une trentaine.
2. Stabilité :
 - Chaque module documenté possède un indice de stabilité dans l'API allant de 0 (module obsolète) à 5 (module verrouillé → le dernier stade où le code ne sera jamais changé).
3. Exemple : le module url → doit être appelé par require (« url »)
 - Les méthodes fournies sont :
 - `parse(url)` qui prend une chaîne URL et retourne un objet URL.
 - `format(urlobj)`, qui fait l'inverse.
 - `resolve(from, to)` qui prend une URL de base et un lien relatif pour construire une URL complète.



Variables globales

1. Node définit les variables suivantes, accessibles dans tous les modules.
2. `global` : représente le contexte racine du fichier courant (un peu comme « window » dans le navigateur).
3. `process` : représente le processus courant.
 - Evénements :
 - `exit` : émis juste avant l'arrêt de Node.
 - `uncaughtException` : émis quand une exception n'a pas été rattrapée.

```
process.on('exit', function() {  
    console.log('Le programme se termine.');//  
})
```

Variables globales

1. Node définit les variables suivantes, accessibles dans tous les modules.
2. `global` : représente le contexte racine du fichier courant (un peu comme « window » dans le navigateur).
3. `process` : représente le processus courant.
 - Evènements :
 - `exit` : émis juste avant l'arrêt de Node.
 - `uncaughtException` : émis quand une exception n'a pas été rattrapée.

```
process.on('uncaughtException',
  function(exception) {
    console.error('Erreur:', exception);
    process.exit(1);
})
```

Process - flux

1. Il y a trois flux disponibles :
 - stdin : flux d'entrée standard,
 - stdout : flux de sortie standard,
 - stderr : flux de d'erreur.
2. Exemple : copie l'entrée standard sur la sortie standard et dans des fichiers.

```
echo 'foo bar' | node test1.js trace.txt
```

Process (suite)

```
// Importe la fonction createWriteStream du module fs
// qui permet d'écrire un fichier via un flux.
var sw = require('fs').createWriteStream();

// Récupère le tableau des arguments passés au programme.
var args = process.argv.slice(2);

// Options de l'écriture du fichier
var opts = {
    flags: args.indexOf('-a') === -1 ? 'w' : 'a'
};

// Transfère le contenu lu depuis l'entrée standard vers la sortie standard
process.stdin.pipe(process.stdout);

// Transfère le contenu de l'entrée standard vers chaque fichier
args.forEach(function(file) {
    if (file.indexOf('.') > -1) // Le nom du fichier doit contenir un .
        process.stdin.pipe(sw(file, opts));
});
```

Process (suite)

1. L'objet process contient également un ensemble d'informations sur le contexte d'exécution.

argv	Vaut Node + nom du script + paramètres
env	Tableau contenant les variables d'environnement
pid	Numéro du processus
arch	Architecture courante (arm, ia32, x64)
platform	Système d'exploitation (darwin, freebsd, Linux, sunos, Win32)
chdir(dossier)	Modifie le dossier courant
cwd()	Retourne le dossier courant
exit([code])	Arrête le processus avec un code d'erreur (optionnel)
getuid()	Retourne et setuid(uid) définit l'utilisateur courant
getgid()	Retourne et setgid(gid) définit le groupe courant

© m2ifformation JVS-NOD 



Métriques

1. Node fournit aussi des métriques, très utiles à la fois dans le processus de développement et en production :
 - `uptime()`, retourne le nombre de secondes depuis le lancement de Node.
 - `hrtime([start])`, retourne le nombre de secondes et nanosecondes écoulées depuis un moment arbitraire dans le passé indépendant du système.
 - `memoryUsage()`, récupère quelques informations sur l'utilisation de la mémoire.



Métriques

1. Node fournit aussi des métriques, très utiles à la fois dans le processus de développement et en production :

```
var start = process.hrtime();
// Faire quelque chose ...
var duration = process.hrtime(start);

console.log('%s secondes et %s nanosecondes', duration[0],
duration[1]);

var usage = process.memoryUsage();

console.log('Utilisation de la memoire :', usage.rss);
console.log('Utilisation du tas %s sur %s :', usage.heapUsed,
usage.heapTotal);
```

Console

1. L'objet console fournit une interface pour afficher des données sur la sortie standard :

- `log([Data...])`, affiche les valeurs sur la sortie standards.
- `error([Data...])`, affiche les valeurs sur la sortie d'erreur.
- `trace([message])`, affiche la pile des appels courante.
- `time(id)` et `timeEnd(id)`, permettent de mesurer facilement le temps d'exécution de morceaux de code.

© m2iformationJVS-NOD

Console

1. L'objet console fournit une interface pour afficher des données sur la sortie standard :

```
console.log({foo: ['bar', 'bat']});
console.log('Valeur %d, %j', 1, 'foo');
console.error('Erreur de lecture');
foo();
function foo() {
    console.trace('Foo');
}
console.time('foo');
// TODO something...
console.timeEnd('foo');
```

© m2ifformation JVS-NOD



51

Console

1. L'objet console fournit une interface pour afficher des données sur la sortie standard :

```
{ foo: [ 'bar', 'bat' ] }
Valeur 1, "foo"
Erreur de lecture
Trace: Foo
  at foo (C:\Users\Serge\Node\page44.js:6:10)
  at Object.<anonymous> (C:\Users\Serge\Node\page44.js:4:1)
  at Module._compile (module.js:413:34)
  at Object.Module._extensions..js (module.js:422:10)
  at Module.load (module.js:357:32)
  at Function.Module._load (module.js:314:12)
  at Function.Module.runMain (module.js:447:10)
  at startup (node.js:141:18)
  at node.js:933:3
foo: 0.308ms
```

© m2iformation JVS-NOD 52



Buffer

1. C'est une classe qui est utilisée pour manipuler des données binaires.
2. Un tampon (buffer) représente une suite d'octets, qui peuvent être vus comme des entiers compris entre 0 et 255.

Buffer

```
***** Creation *****
// Création d'un tampon de 50Ko constitué uniquement d'espaces.
var buf1 = new Buffer(50 * 1e3);
// Création d'un tampon à partir d'un tableau d'entiers.
var buf2 = new Buffer([102, 111, 111, 32, 98, 97, 114]);
// Création d'un tampon à partir d'une chaînes.
var buf3 = new Buffer('foo bar');
// Création d'un tampon à partir d'un encodage.
var buf4 = new Buffer('666f6f20626172', 'hex');

***** Conversion *****
var str = buf2.toString();
var integers = buf3.toJSON();

***** Manipulation *****
console.log(buf2[0]);
buf2[6] = 122;
// Remplit 'z' de l'indice 0 à 3 du buffer
buf2.fill('z', 0, 3);
// Ecrit une chaîne dans le buffer
buf2.write('foo');
```



Divers

1. **require()** : alias de module.require().
2. **__filename et __dirname** : correspondent respectivement au chemin absolu du fichier courant et à son dossier courant.
3. **module** : représente le module courant.
4. **exports** : alias sur module.exports().
5. Minuteurs :
 - setTimeout(callback, ms, [arg...]) et clearTimeout(ref);
 - setInterval(callback, ms, [arg...]) et clearInterval(ref);
 - setImmediate(callback, [arg...]) et clearImmediate(ref).



Programmation asynchrone

1. JavaScript est « mono-threadé ».
2. Problèmes des threads :
 - Les accès aux données partagées doivent être protégés par des verrous,
 - Tout oubli dans le code peut introduire une corruption des données,
 - Un verrouillage mal géré peut mener à des inter blocages,
 - Problèmes de performances (activer, désactiver des verrous est très couteux en termes de ressources),
 - L'utilisation massive de threads peut poser des problèmes de performances.
3. Node se base sur un modèle coopératif et non préemptif : il n'y a pas d'interruption au milieu d'une tâche. Vous éliminez donc les problèmes de synchronisation (pas de verrou) mais vous perdez la concurrence CPU.



API asynchrone dans Node

1. Node utilise le *Continuation-Passing Style* (CPS), c'est-à-dire le passage d'une continuation à une fonction asynchrone pour en recevoir le résultat quand celui-ci est prêt.
2. Syntaxe :

```
asyncOperation(arg1, arg2, function continuation(result){  
    // result est le resultat.  
});
```

3. Exemple :

```
var readFile = require('fs').readFile;  
readFile('test.txt', function(error, result) {  
    if (error) {  
        console.error(error);  
        return;  
    }  
    console.log(result.toString());  
});
```

En informatique, la continuation d'un système désigne son futur, c'est-à-dire la suite des instructions qu'il lui reste à exécuter à un moment précis.

Programmation événementielle

1. Un des paradigmes essentiels de Node pour le code est organisé autour du concept d'évènements : un évènement est associé à un nom, et peut survenir zéro ou plusieurs fois.
2. Tous les objets qui utilisent les évènements dérivent de la classe Event-Emitter du paquet events. Voici les méthodes disponibles :

emit(event[,args...])	Emit un évènement avec un certains nombre de paramètres.
on(event, listener)	Ajoute un listener (écouteur) pour un évènement.
removeListener(event, listener)	Retire un listener d'un évènement.

```
var events = require('events');
var emitter = new events.EventEmitter();
emitter.on('user.create', function onUserCreate(name, age) {
    console.log('Nouvel utilisateur %s (%s ans)', name, age);
});
emitter.emit('user.create', 'Adèle', 43);
```

Boucle d'évènements

1. Fil d'exécution, de type FIFO.
2. Principe :
 - Une callback s'inscrit pour le ou les évènements qui l'intéressent. Ensuite, la boucle d'évènements attend les évènements et lance les abonnées (l'abonné, c'est la callback qui s'est inscrit à l'évènement). Les callbacks ne sont pas interrompues (modèle non préemptif), et elles sont généralement très courtes.

Atelier : concepts



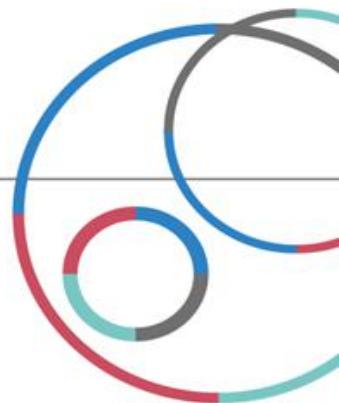
A cartoon illustration of a person with dark hair, wearing a purple shirt, sitting at a desk and working on a vintage-style computer. The computer includes a CRT monitor, a keyboard, and a mouse. An open book lies next to the keyboard. The background is a light blue gradient.

© m2ifformation JVS-NOD

60



Notre expertise est votre avenir



Flux
Eflux



JVS-NOD

Qu'est-ce qu'un flux ?

1. Proche des fameux pipes sous Unix ou Windows.
2. Permet de passer des données tout en autorisant des combinaisons puissantes et flexibles.
3. Plus concrètement, un flux peut être une requête vers un serveur HTTP, ou encore une sortie standard stdout.
4. Il en existe trois types :
 - Flux de lecture (readable)
 - Flux d'écriture (writable)
 - Duplex (les deux à la fois)
 - Il existe également un flux de transformation (transform), qui est un flux de type duplex dans lequel ce qui en sort est une transformation de ce qui y entre.
5. Deux mode de flux : ceux de type objet et ceux de type binaire (pour les fichiers).



© m2ifformation

JVS-NOD

62



Pourquoi utiliser un flux ?

1. Réduction des délais :
 - Il n'est plus nécessaire d'attendre un traitement complet pour effectuer des opérations. C'est le principe même de la chaîne d'assemblage.
2. Réduction de la consommation mémoire :
 - Seuls les segments en cours de traitement occupent la mémoire.
3. Contre-pression (ou back pressure) :
 - Les données sont lues uniquement lorsqu'elles peuvent être traitées, sans surcharger inutilement les ressources (disques ou processeurs).
4. Les flux représentent la philosophie même de Node, avec des briques simples qui sont capables de communiquer facilement entre elles.



Construction : readable

- Il suffit d'instancier un objet de type readable et d'implémenter la méthode `_read()`.

Construction : readable

```
const Readable = require('stream').Readable;
var stream = new Readable({
  objectMode: true // flux de type objet
});
var array = ['foo', 'bar', 'baz'];
var i = 0;
var n = array.length;
stream._read = function(size) {
  // Tant qu'il reste des entrées dans le tableau
  // on tente de les envoyer
  while (i < n) {
    // Si push renvoie false, c'est que le consommateur n'a pas fini de traiter les messages
    // envoyés. Il n'est donc pas nécessaire de continuer pour éviter de saturer la mémoire
    if (!this.push(array[i++])) {
      return;
    }
  }
  // Signale la fin du flux
  this.push(null);
};
stream._read();
console.log(stream._readableState.buffer);
```



Construction : writable

- Il suffit d'instancier un objet de type writable et d'implémenter la méthode _write()

```
const Writable = require('stream').Writable;
var stream = new Writable({
  objectMode: true // flux de type objet
});
stream._write = function(chunk, encoding, next) {
  console.log(chunk);
  // Signale que le message courant a fini d'être traité
  next();
};

stream._write('Bonjour le monde !', null,
  function(){ console.log('ok'); }
);
```



Construction : duplex

1. Il suffit d'instancier un objet de type duplex et d'implémenter les méthodes `_read` et `_write()`.
2. Transform :
 - Une transformée (ou flux de transformation) est un duplex spécialisé, les messages émis relèvent d'une transformation appliquée aux messages envoyés.

Construction : duplex

```
const Transform = require('stream').Transform;
var stream = new Transform({
    objectMode: true // flux de type objet
});
stream._transform = function(chunk, encoding, next) {
    chunk = String(chunk); // Convertit en chaîne si nécessaire
    // On vérifie que chunk n'est pas vide
    if (!chunk.length) {
        // Signale une erreur
        next(new Error('entrée invalide'));
        return;
    }
    // Pousse le ou les nouveaux messages après transformation
    this.push(chunk.toUpperCase());
    // Signale que le message courant a fini d'être traité
    next();
};
stream._transform("Bonjour le monde !", null, function(error){console.log(error)});
console.log(stream._readableState.buffer);
```



Utilisation : lecture

1. Lecture d'un fichier :

- var stream = require('fs').createReadStream('test.txt');

2. Entrée standard du programme :

- var stream = process.stdin;

3. Mode flot : mode le plus simple ; les données arrivent au fur et à mesure de leurs disponibilités avec des évènements **Data** jusqu'à la fin du flux matérialisé par un évènement **end**.

```
stream.on('Data', function(Data){  
    console.log('donnée reçue :', Data);  
});  
stream.on('end', function() {  
    console.log('Le flux s\'est terminé.');//  
});
```



Utilisation : lecture

1. Mode à la demande : dans ce mode, c'est le consommateur qui demande les données, via la méthode read. Il n'est pas pertinent d'appeler read() en permanence dans l'attente de données. Pour éviter ce comportement, on utilise l'évènement readable qui indique que les données sont disponibles.

```
stream.on('readable', function() {  
    console.log('donnée lue :', stream.read());  
});
```

2. Sélection du mode de lecture :

- A la création, un flux est en mode à la demande. Le simple fait de lui ajouter un auditeur pour l'évènement Data le fait basculer en mode « flot ».
- Il est également possible de changer de mode de lecture manuellement avec les méthodes resume() et pause() qui, respectivement, passent le flux en mode « flot » et en mode à la demande.



Utilisation : écriture

1. Pour envoyer des données dans un flux d'écriture, il suffit d'utiliser la méthode write.

```
stream.write('hello');
stream.write('world');
```

2. Pour indiquer la fin du flux, on utilise la méthode end().

```
stream.end();
```

3. Il est important d'indiquer la fin d'un flux : par exemple, dans le cas d'un flux d'écriture du fichier `fs.createWriteStream()`, cela indique à Node de finir l'écriture et de proprement fermer le fichier.



Utilisation : connexion

1. Précédemment, nous avons vu comment manipuler les flux « à la main ». Dans la plupart des cas, il n'est pas nécessaire de procéder ainsi mais seulement de connecter un ensemble de flux pour que le processus s'opère.
2. La méthode qui permet de connecter un flux de lecture à un flux d'écriture est `pipe()`.

```
input
  .pipe(transform1)
  .pipe(transform2)
  .pipe(output);
```

```
npm install git-zlib --save
```

```
var zlib = require('zlib');
var fs = require('fs');
var gzip = zlib.createGzip();
fs.createReadStream('test.txt')
  .pipe(gzip)
  .pipe(fs.createWriteStream('test.txt.gz'));
```

```
npm install git-zlib --save
```



Omniprésence dans Node

1. Les flux sont partout dans Node, ils sont utilisés dans un nombre important de modules.
2. Réseau :
 - Très adaptés aux problématiques réseaux, les flux permettent d'aborder des cas complexes en toute simplicité.
3. Fichier :
 - Les flux peuvent être aussi utilisés pour faire de la copie de fichiers, avec tous les avantages qu'il procurent (contre-pressure notamment).

Atelier : flux



© m2iformation JVS-NOD

74



Notre expertise est votre avenir



Manipulation des fichiers



JVS-NOD

Manipulation de chemin	
1. Tout est dans le module standard path .	
dirname(path)	Retourne la partie du dossier.
basename(path)	Retourne le nom du fichier.
extname(path)	Retourne l'extension du fichier.
isAbsolute(path)	Indique si le path est un chemin absolu.
join(path1, path2...)	Construit un chemin en joignant les morceaux par le séparateur de la plateforme courante.
normalize(path)	Nettoie un chemin en retirant les séparateurs de trop et en supprimant les références au dossier courant (.).
resolve(from, to)	Construit un chemin absolu pour aller de from à to.
relative(from, to)	Construit un chemin relatif pour aller de from à to.

Manipulation de chemin

1. Tout est dans le module standard **path**.

```
var pathLib = require('path');

console.log(pathLib.dirname('C:\\\\Users\\\\Serge\\\\text1.txt '));
console.log(pathLib.basename('C:\\\\Users\\\\Serge\\\\text1.txt '));
console.log(pathLib.dirname('C:\\\\Users\\\\Serge\\\\text1.txt '));
console.log(pathLib.isAbsolute('text1.txt'));
console.log(pathLib.isAbsolute('C:\\\\Users\\\\Serge\\\\Node\\\\text1.txt'));
console.log(pathLib.join(__dirname, 'Data', 'text1.txt'));
console.log(pathLib.normalize('C:\\\\Users\\\\Serge\\\\Node\\\\Data\\\\..\\\\..\\\\text1.txt'));
console.log(pathLib.resolve('C:\\\\Users\\\\Serge\\\\Node', '..\\\\..\\\\text1.txt'));
console.log(pathLib.relative('C:\\\\Users\\\\Serge\\\\Node', 'C:\\\\Users\\\\Serge\\\\text1.txt'));
```

Manipulation de chemin

1. Tout est dans le module standard **path**.

```
C:\Users\Serge
text1.txt
.txt
false
true
C:\Users\Serge\Node\atelier\data\text1.txt
C:\Users\Serge\text1.txt
C:\Users\Serge\text1.txt
..\text1.txt
```

Manipulation de dossiers

1. Dans le module fs :

<code>readdir(path, callback)</code>	Lit le contenu du dossier <i>path</i> .
<code>mkdir(path, callback)</code>	Crée le dossier <i>path</i> (doit être vide).
<code>rmdir(path, callback)</code>	Supprime le dossier <i>path</i> .

2. Pour créer une arborescence de dossiers, le plus simple est d'utiliser le module `mkdirp`. Ce module est calqué sur le même principe que les fonctions précédentes.

3. Pour supprimer un dossier non vide, utilisez le module `rimraf`?

© m2ifformation JVS-NOD  79

m2i Formation

```
var fs = require('fs');
// Lecture d'un dossier
fs.readdir('.', function(error, files) {
    if (error) {
        console.error('Echec de lecture du dossier : ', error);
    } else {
        console.log('fichiers trouvés', files);
    }
});
// Creation d'un dossier
fs.mkdir('./logs', function(error) {
    if (error) {
        console.error('Echec de la creation du dossier : ', error);
    } else {
        console.log('Dossier créé.');
    }
});
// Suppression d'un dossier
fs.rmdir('./logs', function(error) {
    if (error) {
        console.error('Echec de la suppression du dossier : ', error);
    } else {
        console.log('Dossier supprimé.');
    }
});
```

Manipulation de fichiers : métadonnées

1. Dans le module fs :

<code>exists(path, callback)</code>	Teste l'existence d'un fichier (ou un dossier).
<code>stat(path, callback)</code>	Récupère les métadonnées d'un fichier (ou un dossier).
<code>chown(path, uid, gid, callback)</code>	Change le propriétaire et le groupe d'un fichier.
<code>chmod(path, mode, callback)</code>	Change le mode d'un fichier (permissions).
<code>utimes(path, atime, mtime, callback)</code>	Modifie les dates de dernier accès et de modification d'un fichier.
<code>readFile(path, callback)</code>	Lit le contenu d'un fichier.
<code>writeFile(path, content, callback)</code>	Écrit dans un fichier.
<code>appendFile(path, content, callback)</code>	Écrit à la fin du fichier.
<code>truncate(path, len, callback)</code>	Tronque le fichier <i>path</i> à <i>len</i> octets.
<code>rename(oldPath, newPath, callback)</code>	Renomme le fichier <i>oldPath</i> en <i>newPath</i> .
<code>unlink(path, callback)</code>	Supprime le fichier.
<code>createReadStream(path)</code>	Lit le contenu d'un gros fichier.
<code>createWriteStream(path)</code>	Écrit dans un gros fichier.



Surveillance

1. Les fonction `watch(path)`, `watchFile(path)` et `unwatch(path)` permettent de surveiller les changements d'un fichier ou d'un dossier.
2. Ces fonctions n'ont pas un comportement consistant suivant les plateformes.
3. Il est fortement conseillé d'utiliser le module « `gaze` ».

Surveillance

```
var gaze = require('gaze');
gaze('*', { cwd: '.' }, function(error) {
  if (error) {
    console.error('Echec de la mise en surveillance : ', error);
    return;
  }
  this.on('added', function(path){
    console.log('Ajout du fichier', path);
  });
  this.on('deleted', function(path){
    console.log('Suppression du fichier', path);
  });
  this.on('changed', function(path){
    console.log('Modification du fichier', path);
  });
  this.on('renamed', function(oldPath, newPath) {
    console.log('Fichier %s renommé en %s', oldPath, newPath);
  });
});
```



Manipulation de liens symboliques

1. Un lien symbolique est une référence vers un autre fichier ou dossier. La plupart des opérations de lecture ou d'écriture s'appliquent à la destination du lien en lieu et place du lien lui-même.
2. Parfois, le développeur souhaite manipuler le lien lui-même et non le fichier référencé. Node propose plusieurs opérations alternatives ne suivant pas automatiquement les liens :
 - `lchmod(path, mode, callback)`.
 - `lchown(path, uid, gid, callback)`.
 - `lstat(path, callback)`.
 - `symlink(src, dest, callback)` permet de créer un lien symbolique.
 - `readlink(path, callback)` permet de lire la valeur d'un lien symbolique.

Atelier : manipulation des fichiers



A cartoon illustration of a person with dark hair, wearing a purple shirt, sitting at a light blue desk. They are facing a vintage-style computer setup consisting of a CRT monitor on top of a keyboard unit. A mouse is connected by a wire between them. To the left of the computer, an open book lies on the desk. The person's hands are positioned on the keyboard. The background is plain white.

© m2iformation JVS-NOD

85



Notre expertise est votre avenir



Promesses



JVS-NOD



Callbacks vs promesses

1. L'approche traditionnelle avec Node est d'utiliser des callbacks. Ainsi, on ne bloque pas le serveur en attendant l'exécution d'une tâche : celle-ci est lancée et, une fois terminée, elle appelle une fonction.
2. Cette approche souffre de plusieurs maux :
 - Nombreuses imbriques rendant illisible le code.
 - Gestion des erreurs manuelle.
 - Faire attention à la manière dont on appelle une callback : synchrone ou asynchrone (éviter le mélange des paradigmes).
 - Une callback doit être appelée au maximum une fois.



Promesses

1. Une promesse est un objet représentant une valeur qui sera disponible dans le futur.
2. Paradigme de programmation asynchrone.
3. Node ne fournit pas d'implémentation native, il est nécessaire d'utiliser une bibliothèque telle que Bluebird, A+, Q.

```
var Promise = require('bluebird');
var fs = require('fs');
var readFile = Promise.promisify(fs.readFile);
readFile('test1.txt').then(function(content) {
    console.log(content.toString());
}).catch(function(error) {
    console.error(error);
});
```

Création d'une promesse

1. « A la main » :

```
var Promise = require('bluebird');
var fs = require('fs');
var promise = new Promise(function(resolve, reject) {
  fs.readFile('test1.txt', function(error, content) {
    if (error) {
      reject(error);
    } else {
      resolve(content);
    }
  });
}).then(function(content) {
  console.log(content.toString());
}).catch(function(error) {
  console.error(error);
});
```



Création d'une promesse

1. A partir d'une fonction Node :

- Bluebird fournit une méthode pour convertir une fonction Node en fonction renvoyant une promesse.

```
var Promise = require('bluebird');
var fs = require('fs');
var promise = Promise.promisify(fs.readFile);
promise("test1.txt")
  .then(function(content) { console.log(content.toString());
  })
  .catch(function(error) { console.error(error); });
```

```
var promise = Promise.promisifyAll(fs);
fs.readFileAsync("test1.txt", "utf8")
  .then(function(contents) { console.log(contents); })
  .catch(function(error) { console.error(error); });
```

Création d'une promesse

1. A partir d'un évènement :

- Utiliser le paquet event-to-promise.

```
var eventToPromise = require('event-to-promise');
var fs = require('fs');

// Création d'un flux de lecture et d'un flux d'écriture
var input = fs.createReadStream('test3.txt');
var output = fs.createWriteStream('test3b.txt');

// Connexion des deux flux, ce qui revient à faire une copie
input.pipe(output);

// Création d'une promesse attendant l'évènement finish de output
eventToPromise(output, 'finish')
  .then(function() { console.log('La copie est terminée.')} )
  .catch(function(error) { console.error('La copie a échoué :', error);});
```



Intégration avec Node

1. S'il est possible et trivial de créer une fonction renvoyant des promesses à partir d'une fonction utilisant des callbacks Node, il est également possible de faire le contraire.
2. Pour faire ceci, Bluebird propose la méthode « `nodify` » qui transmet le résultat de la promesse courant à la callback.

```
var Promise = require('bluebird');
Promise.promisify(fs.readFile);

function readJSONFile(path, callback) {
    var fs = require('fs');
    var readFile = {
        return readFile(path).then(JSON.parse).nodify(callback);
    };

    readJSONFile('sessions.json', function(error, Data) {
        console.log(Data);
    });
}
```

Intégration avec les générateurs

- Il est encore plus simple d'utiliser les promesses avec la notion de générateur. Dans un générateur, le mot-clé **yield** permet de retourner une valeur intermédiaire et de suspendre temporairement l'exécution de la fonction.

```
function *range(a, b) {
  var a = Math.ceil(a);
  while (a < b) {
    yield(a);
    a++;
  }
}

var iterator = range(0, 2);
console.log(iterator.next());
console.log(iterator.next());
console.log(iterator.next());
```

```
C:\Users\Serge\Node>node page90b.js
{ value: 0, done: false }
{ value: 1, done: false }
{ value: undefined, done: true }
```



Intégration avec les générateurs

1. Utilisation des générateurs pour réaliser simplement des actions en parallèle, comme la lecture des fichiers.

```
var Promise = require('bluebird');
var fs = require('fs');
var readFile = Promise.promisify(fs.readFile);
var read = Promise.coroutine(function *() {
    try {
        var promises = [ readFile('test1.txt'), readFile('test2.txt') ];
        var contents = yield Promise.all(promises);
        console.log('Contenu de test1.txt : ', contents[0].toString());
        console.log();
        console.log('Contenu de test2.txt : ', contents[1].toString());
    }
    catch (error) { console.error(error); }
});

read();
```

Intégration avec les générateurs

- Bluebird permet d'étendre très facilement ce comportement avec la fonction BlueBird.coroutine.addYieldHandler.

```
Promise.coroutine.addYieldHandler(function arrayHandler(value){  
    if (Array.isArray(value)){  
        return Promise.all(value);  
    }  
});  
var read = Promise.coroutine(function *() {  
    try {  
        var promises = [ readFile('test1.txt'), readFile('test2.txt') ];  
        var contents = yield promises;  
        console.log('Contenu de test1.txt : ', contents[0].toString());  
        console.log();  
        console.log('Contenu de test2.txt : ', contents[1].toString());  
    }  
    catch (error) { console.error(error); }  
});
```

Atelier : promesses



A cartoon illustration of a person with dark hair, wearing a purple shirt, sitting at a light blue desk. They are facing a vintage-style computer monitor and keyboard. An open book lies next to the keyboard. The monitor has small black lines above it, suggesting sound or output. The background is plain white.

© m2iformation JVS-NOD 96



Notre expertise est votre avenir



Application en ligne de commande



JVS-NOD



Introduction

1. Réaliser une application en ligne de commande est extrêmement simple. Il suffit de mettre le code dans un fichier JS et de l'exécuter avec node : **node index.js**
2. Mais la réalisation d'un véritable projet en ligne de commande implique d'autres difficultés comme l'exécution directe (sans préciser Node) ou même la prise en compte de paramètres.



Gestion des paramètres

1. Accessible via la propriété argv de l'objet process,
2. Soit le script contenant le code suivant : `console.log(process.argv);`
3. Soit la ligne suivante exécutant le script précédent :
 - `node slide90.js --foo bar --test`
4. Le script retournera :

```
C:\Users\Serge\Node>node slide90.js --foo bar --test
[ 'C:\\\\Program Files\\\\nodejs\\\\node.exe',
  'C:\\\\Users\\\\Serge\\\\Node\\\\slide90.js',
  '--foo',
  'bar',
  '--test' ]
```



Gestion des paramètres

1. La première étape est de ne sélectionner que les paramètres :
 - `console.log(process.argv.slice(2));`
2. Une fois les paramètres sélectionnés, il est nécessaire de les analyser afin de gérer les alias (par exemple, `-h` et `--help`), les valeurs par défauts...
3. Pour ce travail, on utilise le paquet `minimist`: `npm install --save minimist`.

```
var minimist = require('minimist');
var opts =
minimist(process.argv.slice(2), {
    boolean : ['help'],
    string : ['port'],
    default: {
        port: '80'      C:\Users\Serge\Node>node slide91.js --foo bar --tests
    }                      { _: [], help: false, foo: 'bar', tests: true, port: '80' }
})
console.log(opts);
```



Testabilité

1. Dans l'idéal, un programme en ligne de commande devrait pouvoir être testé de la même façon qu'une bibliothèque.
2. Pour cela, nous devons installer mocha et chai (voir chapitre 11) :
 - `npm install -g mocha`
 - `npm install --save chai`

Testabilité : script à tester

Le script à tester : mon-serveur.js

```
var minimist = require('minimist');
var http = require('http');
var eventPromise = require('event-to-promise')
function main(args) {
    var opts = minimist(args, { boolean : ['help'], string : ['port'], default: { port: '80' } });
    if (opts.help) { return 'Usage: mon-serveur [-port=<port>] <dossier>...'; }
    // Initialisation d'express et du serveur Web
    var server = http.createServer(function(req, res) {
        res.writeHead(200, {"Content-Type": "text/html"});
        res.end('<h1>Salut tout le monde ! </h1>');
    });
    server.listen(opts.port);
    // Pour l'exemple, le serveur se fermera automatiquement au bout d'une heure
    setTimeout(function() {
        server.close();
    }, 1*60*60*1000);
    // Retourne une promesse qui sera résolue quand le serveur se fermera
    return eventPromise(server, 'close');
}
module.exports = main;
// Si le module courant n'a pas été inclus mais est exécuté directement alors, la fonction principale est exécutée par le
// module 'exec-promise'.
if (!module.parent) {
    var execPromise = require('exec-promise');
    execPromise(main);
}
```



Testabilité : script de test

1. Le script de test : `mon-serveur.spec.js`

```
// Pour bien comprendre la structure de ce module, voir le chapitre 10 sur les tests.  
var serveur = require('./mon-serveur');  
var expect = require('chai').expect;  
  
describe('mon-serveur()', function () {  
    it('returns the usage when --help is passed', function () {  
        expect(serveur(['--help'])).to.equal('Usage: mon-serveur [--port=<port>]  
<dossier>...');  
    });  
});
```

- L'exécution du test : `mocha mon-serveur.spec.js`

```
C:\Users\Serge\Node>mocha mon-serveur.spec.js  
  
  mon-serveur()  
    ✓ returns the usage when --help is passed  
  
      1 passing (16ms)
```



Exécution directe

1. Jusqu'ici, le module est exécutable via la commande node.
2. Mais on souhaite l'utiliser directement.
3. Les SE basés sur UNIX permettent de spécifier le chemin de l'interprète (ici node) à utiliser pour le script dans l'en-tête de ce dernier grâce à une syntaxe spéciale appelée **shebang**. Celle-ci consiste à écrire sur la toute première ligne du script **#!** Suivi du chemin absolu de l'interpréteur de commande : **#!/usr/bin/node**.
4. Sous Windows, npm embarque une couche de compatibilité qui rend le programme exécutable quand il est installé via son paquet (voir leçon suivante).

Installation du programme

- Dernière étape de la création du programme, son installation via npm.

```
{  
  "private": true,  
  "name" : "mon-serveur",  
  "version": "0.0.0",  
  "bin": {  
    "mon-serveur": "mon-serveur.js"  
  },  
  "dependencies": {  
    "http": "^0.0.0",  
    "minimist": "^1.2.0",  
    "event-to-promise": "^0.7.0",  
    "exec-promise": "^0.6.1"  
  }  
}
```

package.json

- Tout d'abord, il faut créer une description minimale du paquet dans le fichier package.json.
- On peut maintenant installer le programme et l'exécuter:
 - npm install --global
 - mon-serveur --help

m²i Atelier : application en ligne de commande



A cartoon illustration of a person with dark hair, wearing a purple shirt, sitting at a desk and working on a vintage-style computer. The computer consists of a CRT monitor on top of a keyboard unit. To the left of the computer is an open book. The background is light blue.

© m2ifformation JVS-NOD 106



Notre expertise est votre avenir



Application Web



JVS-NOD

Construire son serveur HTTP

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

var server = http.createServer(function(req, res) {
    res.writeHead(200, {"Content-Type": "text/html"});
    res.end('<h1>Salut tout le monde ! </h1>');
});

server.listen(8080);
```

Construire sa page

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

var server = http.createServer(function(req, res) {
  res.writeHead(200, {"Content-Type": "text/html"});
  res.write('<!DOCTYPE html>\n'+
    '<html>\n'+
    '  <head>\n'+
    '    <meta charset="utf-8" />\n'+
    '    <title>Ma page Node.js !</title>\n'+
    '  </head>\n'+
    '  <body>\n'+
    '    <p>Voici un paragraphe <strong>HTML</strong> !</p>\n'+
    '  </body>\n'+
    '</html>\n');
  res.end();
});

server.listen(8080);
```

Construire sa page

Waaah ! Mais c'est atroce
d'écrire du HTML comme ça !



Aucun développeur ne s'amusera vraiment à faire des pages Web HTML complexes ainsi. Il existe des moyens de séparer le code HTML du code JavaScript : ce sont les systèmes de templates.



Quelle est la page demandée par le visiteur ?

1. Pour récupérer la page demandée par le visiteur, on va faire appel à un nouveau module de Node appelé "url". On demande son inclusion avec :
 - `var url = require("url");`

2. Ensuite, il nous suffit de "parser" la requête du visiteur comme ceci pour obtenir le nom de la page demandée :
 - `url.parse(req.url).pathname;`

Quelle est la page demandée par le visiteur ?

```
var http = require('http');
var url = require('url');

var server = http.createServer(function(req, res) {
    var page = url.parse(req.url).pathname;
    console.log(page);
    res.writeHead(200, {"Content-Type": "text/plain"});
    if (page == '/') {
        res.write('Vous êtes à l\'accueil, que puis-je pour vous ?');
    } else if (page == '/sous-sol') {
        res.write('Vous êtes dans la cave à vins, ces bouteilles sont à moi !');
    } else if (page == '/etage/1/chambre') {
        res.write('Hé ho, c\'est privé ici !');
    }
    res.end();
});
server.listen(8080);
```



Quels sont les paramètres ?

1. Les paramètres sont envoyés à la fin de l'URL, après le chemin du fichier. Prenez cette URL par exemple :
 - `http://localhost:8080/page?prenom=Robert&nom=Dupont`
2. Les paramètres sont contenus dans la chaîne `? prenom=Robert&nom=Dupont`. Pour récupérer cette chaîne, il suffit de faire appel à :
 - `url.parse(req.url).query`
3. Le problème, c'est qu'on vous renvoie toute la chaîne sans découper au préalable les différents paramètres. Heureusement, il existe un module Node.js qui s'en charge pour nous : `querystring` !
 - `var querystring = require('querystring');`
4. Vous pourrez ensuite faire :
 - `var params = querystring.parse(url.parse(req.url).query);`

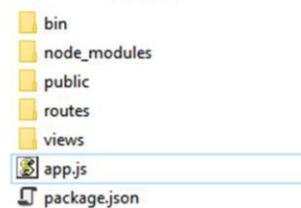
Quels sont les paramètres ?

```
var http = require('http');
var url = require('url');
var querystring = require('querystring');

var server = http.createServer(function(req, res) {
    var params = querystring.parse(url.parse(req.url).query);
    res.writeHead(200, {"Content-Type": "text/plain"});
    if ('prenom' in params && 'nom' in params) {
        res.write('Vous vous appelez ' + params['prenom'] + ' ' +
                  params['nom']);
    } else {
        res.write('Vous devez bien avoir un prénom et un nom, non ?');
    }
    res.end();
});
server.listen(8080);
```

Générateur Express

1. Dans ce chapitre, nous allons nous intéresser à Express (<http://expressjs.com/>), un framework Web minimaliste, simple mais néanmoins puissant.
2. Tout d'abord, l'installer : `npm install --global express-generator`
3. Créer un dossier pour le site Web et s'y rendre puis lancer sa création :
 - a. Express,
 - b. Installer les dépendances : `install dependencies`,
 - c. Modifier le fichier `app.js` et ajouter `app.listen(80)` pour le faire écouter le port 80 (par exemple),
 - d. Exécuter l'application : `node app.js` puis aller sur le navigateur : `http://localhost`.





Architecture

1. L'architecture d'Express est basée sur l'exécution en cascade de middleware à chaque requête entrante.
2. Un middleware pour Express est une fonction qui reçoit en paramètre la requête et la réponse courantes ainsi que le middleware suivant :

```
function foo(request, response, next) {
    // Affiche la requête courante
    console.log('Requête : ', request.method, ' sur ', request.path);
    // Passe la main au middleware suivant
    next();
}
```

3. L'association d'un middleware à l'application se fait via la méthode use :

```
app.use(foo);
```

Requête

<http://expressjs.com/fr/4x/api.html#req>

Propriété	Description
body	Tableau associatif des données soumises d'un formulaire.
cookies	Objet contenant les cookies.
method	Méthode d'envoi de la requête (GET ou POST).
path	URL de la requête.
params	Paramètres de la route.
query	Cette propriété est un objet contenant une propriété pour chaque paramètre dans la route.
get(field)	Retourne la valeur de la propriété d'entête HTTP spécifiée.

© m2iformation JVS-NOD  117

Réponse	
http://expressjs.com/fr/4x/api.html#res	
Propriété	Description
headerSent	Booléen indiquant si l'en-tête HTTP a déjà été envoyé.
append(field [, value])	Modifie la valeur d'une propriété d'en-tête HTTP.
attachment([filename])	Envoie le fichier spécifié dans la réponse.
cookie(name, value [, options])	Crée un cookie.
json([body])	Envoie un contenu JSON.
location(path)	Modifie la propriété location du header, redirige vers la page spécifiée.
redirect([status,] path)	Redirige vers la page spécifiée avec un code d'état HTTP.
render(view [, locals] [, callback])	Renvoie vers la vue spécifiée avec des paramètres (locals).
send([body])	Envoie la réponse HTTP.

Routeur

<http://expressjs.com/fr/4x/api.html#router>

Propriété	Description
all	Toutes méthodes (GET, POST)
get	Méthode GET
post	Méthode POST
route	Retourne une instance d'une seule route que vous pouvez ensuite utiliser pour gérer les verbes HTTP (GET, POST, PUT, DELETE).
use([path], [function, ...] function)	Utilise la ou les fonctions middleware spécifiée(s) , avec le chemin path en option (par défaut "/").



Distribution de fichiers statiques

1. Pour se servir des fichiers statiques, nul besoin d'écrire son propre middleware. En effet, Express embarque le module serve-static qui est dédié à cet usage.
2. Son utilisation est triviale, il suffit d'appeler la fonction express.static() en lui passant en paramètre le chemin du dossier contenant les fichiers à servir.

```
app.use(express.static(__dirname + '/public'));
```



Le moteur de rendu EJS

1. EJS (Embedded Javascript) est un moteur de rendu permettant de générer un document HTML à partir de template type ASP (comme en PHP, J2EE...).
2. Les tags :
 - `<%` Tag 'Scriptlet', pour le contrôle de flux, pas de sortie
 - `<%=` Affiche la valeur du modèle - désactive les tags HTML inclus
 - `<%-` Affiche la valeur du modèle - ne désactive pas les tags HTML inclus
 - `<%#` Tag 'Commentaire'



Création d'un helper

1. Le but est de faciliter l'écriture de données dans le fichier ejs.
2. Exemple : une liste déroulante :

```
<select class="form-control" name="maliste">
  <% liste.forEach(function(item, index) { %>
    <option value="<%= item.value %>"><%= item.text %></option>
  <% }); %>
</select>
```

3. Pourrait s'écrire :

```
<%- listBox('maliste', liste, 'value', 'text') %>
```



Création d'un helper

1. Le module helpers

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

exports.helpers = {
  listBox: function(name, liste, value, text) {
    if (typeof(value) == 'undefined') value = 'value';
    if (typeof(text) == 'undefined') text = 'text';
    var content = '<select name="' + name + '" class="form-control">';
    liste.forEach(function(item, index) {
      content += '<option value="' + item[value] + '">';
      content += item[text] + '</option>';
    });
    content += '</select>';
    return ejs.render(content);
  }
};
```

2. Dans app.js

```
app.locals = require('./modules/helpers.js').helpers;
```



Le moteur de rendu Jade

1. Jade : <http://jade-lang.com/reference/>

```
!!! 5
html
  head
    title Example
  body Users :
    ul
      for user in users
        li #{user.name} : #{user.age}
```



Le moteur de rendu Mustashe

- Mustashe : <https://mustache.github.io/>

```
<!DOCTYPE html>
<html>
<head>
</head>
<body>
  {{>homepage}}
</body>
```

master.html

```
<h1>{{homepage.title}}</h1>
<ul>
{{#homepage.items}}
  <li><a href="posts/{{_id}}">{{title}}</a></li>
{{/homepage.items}}
</ul>
```

homepage.html

Exemple

1. Gestion d'une liste de tâches

Ma todolist

- X Le café
- X Le repassage
- X Préparer le repas

Que dois-je faire ? Valider



© m2iformation JVS-NOD

126



Le fichier package.json

```
{  
  "name": "todo",  
  "version": "0.1.0",  
  "private": true,  
  "scripts": {  
    "start": "node ./bin/www"  
  },  
  "dependencies": {  
    "express": "~4.13.1",  
    "ejs": "~2.1.4",  
    "body-parser": "~1.13.2",  
    "cookie-session": "~1.1.0"  
  },  
  "author": "Serge <s.lhostis@m2iformation.fr>",  
  "description": "Un gestionnaire de todolist ultra basique"  
}
```



Le fichier app.js

```
var express = require('express');
var path = require('path');
var ejs = require('ejs');
var session = require('cookie-session'); // Charge le middleware de sessions
var bodyParser = require('body-parser'); // Charge le middleware de gestion des paramètres
var urlencodedParser = bodyParser.urlencoded({ extended: false });

var app = express();
var todo = require('./routes/todo'); // Le contrôleur
app.set('views', path.join(__dirname, 'views')); // Le chemin de la vue

app.use(session({ secret: 'todotopsecret' })); // Utilisation des sessions
app.set("view options", { layout: false });
app.engine('html', ejs.renderFile);
app.set('view engine', 'html');
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));
app.use(express.static(path.join(__dirname, 'public')));

// Pas de todolist dans la session, créer une vide
app.use(function (req, res, next) { if (!req.session.todolist) { req.session.todolist = []; }; next(); });

app.use('/todo', todo); // La route
// Rediriger vers la todolist si la page demandée n'est pas trouvée
app.use(function (req, res, next) { res.redirect('/todo'); });

app.listen(8080);
module.exports = app;
```

Le contrôleur todo.js

```
var express = require('express');
var session = require('cookie-session'); // Charge le middleware de sessions
var router = express.Router();
var bodyParser = require('body-parser'); // Charge le middleware de gestion des paramètres
var urlencodedParser = bodyParser.urlencoded({ extended: false });

/* Page d'accueil */
router.get('/', function(req, res, next) {
  res.render('todo.ejs', { title: 'Express', todolist: req.session.todolist});
});

/* Ajoute d'un élément à la todolist */
router.post('/ajouter/', urlencodedParser, function(req, res) {
  if (req.body.newtodo != "") {
    req.session.todolist.push(req.body.newtodo);
  }
  res.redirect('/');
});

/* Suppression d'un élément de la todolist */
router.get('/supprimer/:id', function(req, res) {
  if (req.params.id != "") {
    req.session.todolist.splice(req.params.id, 1);
  }
  res.redirect('/');
});

module.exports = router;
```

La vue todo.ejs

```
<!DOCTYPE html>
<html>
  <head>
    <title>Ma todolist</title>
    <style>
      a {text-decoration: none; color: black;}
    </style>
  </head>
  <body>
    <h1>Ma todolist</h1>

    <ul>
      <% todolist.forEach(function(todo, index) { %>
        <li><a href="/todo/supprimer/<%= index %>">X </a> <%= todo %></li>
      <% }); %>
    </ul>
    <form action="/todo/ajouter/" method="post">
      <p>
        <label for="newtodo">Que dois-je faire ?</label>
        <input type="text" name="newtodo" id="newtodo" autofocus />
        <input type="submit" />
      </p>
    </form>
  </body>
</html>
```

Atelier : application Web



A cartoon illustration of a person with dark hair, wearing a purple shirt, sitting at a desk and working on a vintage-style computer. The computer includes a CRT monitor, a keyboard, and a mouse. An open book lies next to the keyboard. The background is a light blue gradient.

© m2ifformation JVS-NOD

131



Notre expertise est votre avenir



Base de données

Base de données



JVS-NOD

Connexion à MySQL

1. npm install mysql

```
var mysql = require('mysql');
var cn = mysql.createConnection({
    host      : 'localhost',
    user      : 'root',
    password  : '',
    Database  : 'formationsM2i'
});

cn.connect();
cn.query('SELECT * from client', function(err, rows, fields) {
    if (err) throw err;
    for (var i=0; i<rows.length; i++) {
        console.log(rows[i].RaisonSociale);
    }
});
cn.end();
```

Connexion à MySQL

1. Pool de connexions

```
var pool = mysql.createPool({  
  connectionLimit : 100, //important  
  host      : 'localhost',  
  user      : 'root',  
  password   : '',  
  Database  : 'tododb',  
  debug      : false  
});
```

Connexion à MySQL

```
pool.getConnection( function(err, connection) {
  if (err) {
    connection.release();
    console.log({ "code" : 100, "status" : "Error in connection Database"});
    return;
  }
  console.log('connected as id ' + connection.threadId);
  var query = connection.query("SELECT id, label FROM todo", function(err, rows){
    connection.release();
    if (!err) {
      res.render('todos.ejs', { title: 'Gestion des todos', todoList: rows });
    }
  });
  connection.on('error', function(err) {
    console.log({ "code" : 100, "status" : "Error in connection Database"});
    return;
  });
});
```

Passage de paramètres

1. Paramètres simple :

```
connection.query("SELECT * FROM table WHERE id=? LIMIT ?, 5",
  [id, start],
  function (err, result) { ... });
```

2. Tableau de paramètres

```
var columns = ['id', 'label'];
connection.query("SELECT ?? FROM table WHERE id=? LIMIT ?, 5",
  [columns, id, start],
  function (err, result) { ... });
```



Passage de paramètres

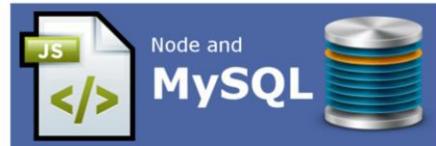
1. Format personnalisé

```
connection.config.queryFormat = function (query, values) {  
    if (!values) return query;  
    return query.replace(/\:(\w+)/g, function (text, key) {  
        if (values.hasOwnProperty(key)) {  
            return this.escape(values[key]);  
        }  
        return text;  
    }.bind(this));  
};  
  
connection.query("UPDATE posts SET title = :title",  
    { title: "Hello MySQL" });
```



Création d'un module d'accès aux données

1. Exemple :



Création d'un module d'accès aux données

1. Dans un dossier « modules ». Le fichier se nommera db.js :

```
var mysql = require('mysql');
var Promise = require('bluebird');

/* Constructeur */
var db = function () {
    this.pool = mysql.createPool({
        connectionLimit : 100, //important
        host            : 'localhost',
        user            : 'root',
        password        : '',
        Database        : 'myDatabase',
        debug           : false
    });
}
```

Le module

```
db.prototype.get = function() {
    var self = this; // Pour qu'il soit accessible dans la Promise
    return new Promise(function(resolve, reject) {
        self.pool.getConnection(function(err, connection) {
            if (err !== null) {
                connection.release();
                reject({ "code" : 100, "status" : "Error in connection Database" });
                return;
            }
            var query = connection.query("SELECT * FROM table", function(err, rows){
                connection.release();
                if (err === null) { resolve(rows); } else { reject(err); }
            });
            connection.on('error', function(err) {
                reject({ "code" : 100, "status" : "Error in connection Database" });
                return;
            });
        });
    }).then(function(Data) { return Data;
    }).catch(function(error) { throw (error); });
};

exports.db = db;
```

Le routeur

1. Dans le dossier « routes ». Le fichier se nommera index.js :

```
var express = require('express');
var dbModule = require('../modules/db'); //On ne met pas l'extension

var router = express.Router();
var db = new dbModule.db();

router.get('/', function(req, res, next) {
    db.get()
        .then(function(Data) {
            res.render('mapage.ejs', { title: 'Mon titre',
                                      list: Data });
        })
        .catch(function(error) {
            res.render('erreur.ejs', { title: 'Erreur',
                                      errorMessage: error });
        });
});
module.exports = router;
```

La vue

1. Dans le dossier « views ». Le fichier se nommera index.ejs :

```
<!DOCTYPE HTML>
<html>
  <head>
    <title> Ma Vue </title>
    <meta charset="UTF-8">
  </head>
  <body>
    <h1><%= title %></h1>
    <ul>
      <% list.forEach(function(item, index) { %>
        <li><%= item.text %></li>
        <% }); %>
      </ul>
    </body>
  </html>
```

© m2iformation JVS-NOD  142

Atelier : base de données



A cartoon illustration of a person with dark hair, wearing a purple shirt, sitting at a desk and working on a vintage-style computer. The computer includes a CRT monitor, a keyboard, and a mouse. An open book lies next to the keyboard. The background is a light blue gradient.

© m2ifformation JVS-NOD 143



Notre expertise est votre avenir



Socket.IO



JVS-NOD

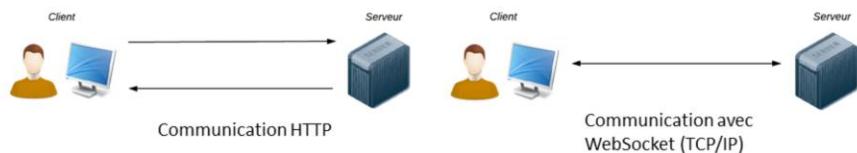


Introduction

1. Socket.IO est l'une des bibliothèques les plus prisées par ceux qui développent avec Node.js.
2. Elle permet de faire très simplement de la communication synchrone dans votre application, c'est-à-dire de la communication en temps réel !
3. Elle se base sur plusieurs techniques différentes qui permettent la communication en temps réel. La plus connue d'entre elles, et la plus récente, est WebSocket (API HTML5).

WebSocket

1. WebSocket est une fonctionnalité supportée par l'ensemble des navigateurs récents.
Elle permet un **échange bilatéral synchrone** entre le client et le serveur.



Ne confondez pas WebSocket et Ajax!

Ajax permet, effectivement, au client et au serveur d'échanger des informations sans recharger la page. Mais en Ajax, c'est toujours le client qui demande et le serveur qui répond. Le serveur ne peut pas décider de lui-même d'envoyer des informations au client. Avec WebSocket, ça devient possible !

© m2ifformation JVS-NOD

146

Socket.IO

1. Installation : via un fichier JSON : `package.json`

```
{  
  "name": "chat",  
  "version": "0.1.0",  
  "dependencies": {  
    "express": "~3.3.4",  
    "Socket.IO": "~1.4.5",  
    "ent": "~0.1.0"  
  },  
  "author": "Serge <s.lhostis@m2iformation.fr>",  
  "description": "Un Chat temps réel avec Socket.IO"  
}
```

2. Installer : `npm install`

© m2iformation JVS-NOD  147

Socket.IO : le fichier app.js

```
var http = require('http');
var fs = require('fs');
// Chargement du fichier index.html affiché au client
var server = http.createServer(function(req, res) {
    fs.readFile('./index.html', 'utf-8', function(error, content) {
        res.writeHead(200, {"Content-Type": "text/html"});
        res.end(content);
    });
});
// Chargement de Socket.IO
var io = require('Socket.IO').listen(server);
// Quand un client se connecte, on le note dans la console
io.sockets.on('connection', function (socket) {
    console.log('Un client est connecté !');
});
server.listen(8080);
```

Socket.IO : le fichier index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>Socket.IO</title>
    <script src=".js/Socket.IO.js"></script>
  </head>

  <body>
    <h1>Communication avec Socket.IO !</h1>
    <script>
      var socket = io.connect('http://localhost:8080');
    </script>
  </body>
</html>
```

m2i
Formation

Exécution de l'application

1. node app.js

Un client est connecté !

© m2iformation JVS-NOD 

Envoi et réception d'un message

1. Dans app.js (à la place du code de l'évènement connexion) :

```
io.sockets.on('connection', function (socket) {
    socket.emit('message', 'Vous êtes bien connecté !');
});
```

2. Dans index.html :

```
<script>
var socket = io.connect('http://localhost:8080');
socket.on('message', function(message) {
    alert('Le serveur a un message pour vous : ' + message);
})
</script>
```



Le client envoi un message

1. Dans index.html, référencer jQuery :

```
<script src="http://code.jquery.com/jquery-1.10.1.min.js"></script>
```

- Puis :

```
<p><input type="button" value="Embêter le serveur" id="poke" /></p>
<script>
    var socket = io.connect('http://localhost:8080');
    socket.on('message', function(message) {
        alert('Le serveur a un message pour vous : ' + message);
    });
    $('#poke').click(function () {
        socket.emit('message', 'Salut serveur, ça va ?');
    });
</script>
```



Réception du message du client

1. Dans app.js :

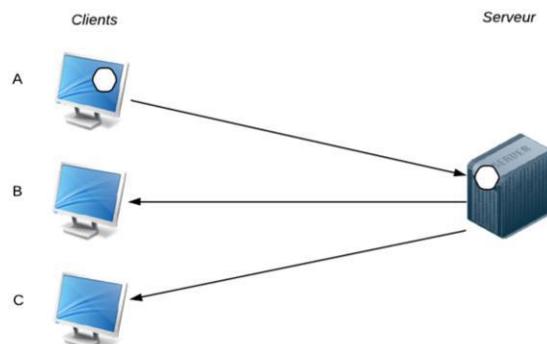
```
io.sockets.on('connection', function (socket) {
    socket.emit('message', 'Vous êtes bien connecté !');

    // Quand le serveur reçoit un signal de type "message" du client
    socket.on('message', function (message) {
        console.log('Un client me parle ! Il me dit : ' + message);
    });
});
```



Communiquer avec plusieurs clients

1. Le client A envoie un message au serveur.
2. Le serveur l'analyse.
3. Il décide de broadcaster ce message pour l'envoyer aux autres clients connectés : B et C.



© m2iformation

JVS-NOD

154



Envoi d'un message à tous les clients

1. Dans app.js :

```
io.sockets.on('connection', function (socket) {  
    socket.emit('message', 'Vous êtes bien connecté !');  
    socket.broadcast.emit('message',  
        'Un autre client vient de se connecter !');  
});
```

Les variables de session

1. Lorsque vous aurez plusieurs clients connectés, vous allez vite vous rendre compte qu'il est délicat de les reconnaître. L'idéal serait de pouvoir mémoriser des informations sur chaque client sous forme de variables de session... Mais, par défaut, Socket.IO ne propose pas cette fonctionnalité.
2. En fait, les variables de sessions doivent être gérées par une bibliothèque supplémentaire sous forme de *middleware* comme [session.Socket.IO](#).
 - Côté client :

```
var pseudo = prompt('Quel est votre pseudo ?');
socket.emit('username', pseudo);
```

- Côté serveur :

```
socket.on('username', function(pseudo) {
  socket.pseudo = pseudo;
});
```

Le script (presque) complet : le client

```
<script>
  var socket = io.connect('http://localhost:8080');

  // On demande le pseudo au visiteur...
  var pseudo = prompt('Quel est votre pseudo ?');
  // Et on l'envoie avec le signal "username" (pour le différencier de
  "message")
  socket.emit('username', pseudo);

  // On affiche une boîte de dialogue quand le serveur nous envoie un "message"
  socket.on('message', function (message) {
    alert('Le serveur a un message pour vous : ' + message);
  });

  // Lorsqu'on clique sur le bouton, on envoie un "message" au serveur
  $('#poke').click(function () {
    socket.emit('message', 'Salut serveur, ça va ?');
  });
</script>
```

Le script (presque) complet : le serveur

```
// Chargement de Socket.IO
var io = require('Socket.IO').listen(server);

io.sockets.on('connection', function(socket, pseudo) {
    // Quand un client se connecte, on lui envoie un message
    socket.emit('message', 'Vous êtes bien connecté !');
    // On signale aux autres clients qu'il y a un nouveau venu
    socket.broadcast.emit('message', 'Un autre client vient de se connecter !');
    // Dès qu'on nous donne un pseudo, on le stocke en variable de session
    socket.on('username', function(pseudo) {
        socket.pseudo = pseudo;
    });
    // Dès qu'on reçoit un "message" (clic sur le bouton), on le note dans la
    console
    socket.on('message', function(message) {
        // On récupère le pseudo de celui qui a cliqué dans les variables de
        session
        console.log(socket.pseudo + ' me parle ! Il me dit : ' + message);
    });
});
```

Atelier : Socket.IO



A cartoon illustration of a person with dark hair, wearing a purple shirt, sitting at a light blue desk. They are facing a vintage-style computer setup consisting of a CRT monitor on top of a keyboard unit. A mouse sits on the desk to the right of the keyboard. An open book lies on the desk to the left of the computer. The person is looking at the screen, with small black lines radiating from it to indicate they are interacting with the computer.

© m2ifformation JVS-NOD 159



Notre expertise est votre avenir



Tests



JVS-NOD



Introduction

1. La bonne qualité des logiciels passe par des tests :
 - Comment être certain qu'un livrable se comporte comme prévu ?
 - Est-ce que la nouvelle fonctionnalité introduite n'en casse pas d'autres ? (Test de régression.)
2. Rendre un code testable, c'est gagner du temps par la suite.
3. Le test unitaire est une procédure permettant de vérifier le bon fonctionnement d'une partie précise d'un logiciel.
4. Lesdits test peuvent représenter un nombre de lignes conséquent. Cependant, les chiffres sont à mettre en perspective :
 - Combien de temps passe-t-on à l'écriture du code par rapport au temps dédié à sa maintenance et à son évolutivité ?



Lanceur de tests - Mocha

1. Mocha est un des lanceurs de tests les plus utilisés.
2. Mocha permet :
 - Exécution des tests avec Node,
 - Exécution des tests dans un navigateur,
 - Support des promesses,
 - Génération de rapports sur la couverture des tests,
 - Facilité de génération des cas de tests,
 - Vérification des tests lents...



Mocha - Installation

1. Installation de mocha : `npm install -g mocha`
2. Modification de package de l'application :
3. Ecriture de la série de tests dans index.spec.js
4. Lancement du test : [npm test](#)

```
{  
  ...  
  "private":true,  
  "scripts": {  
    "start": "node app.js",  
    "test": "mocha index.spec.js"  
  },  
  "dependencies":{ ... },  
  "devDependencies":{  
    "mocha": "*"  
  },  
  ...  
}
```

Création d'une suite de tests

1. Mocha propose plusieurs interfaces de programmation pour écrire des jeux de tests.
2. Ici, nous utiliserons BDD (*Behavior Driven Development*).
3. La bonne pratique veut que la concaténation des descriptions dans la hiérarchie des `describe()` et des `it()` forment une phrase compréhensible en anglais.

```
describe('Array' , function() {  
    describe('.length', function() {  
        it('returns the number of items');  
        it('can be used to change the size');  
    });  
});
```

Exemple

```
module.exports = function() {
  Array.prototype.first = function() {
    return this[0];
  };
  Array.prototype.last = function() {
    return this[this.length - 1];
  };
}
```

array.js

```
var assert = require('assert');
require('../js/array.js');

describe('Array#prototype', function() {
  it('should return the first element of my array',
    function() {
      assert.equal([1, 3, 3, 4, 5].first(), 1);
    });
  it('should return the last element of my array',
    function() {
      assert.equal([1, 3, 3, 4, 5].last(), 5);
    });
});
```

test.js

```
formationNJS@0.1.0 test c:\wamp\www\formationANG
mocha ./test/test.js
```

```
Array#prototype
  ✓ should return the first element of my array
  ✓ should return the last element of my array

2 passing (16ms)
```

Résultat

© m2ifformation

JVS-NOD



165

Mocha

1. Les fonctions de Mocha :

Fonctions	Description
describe	Définit une suite de tests qui contient des cas de test
it	Décrit un cas de test
before	Décrit une phase d'exécution avant les tests
after	Décrit une phase d'exécution après les tests
beforeEach	Décrit une phase d'exécution avant chaque test
afterEach	Décrit une phase d'exécution après chaque test
assert	Teste une assertion. Nécessite de charger le module assert de Mocha

© m2ifformation JVS-NOD  166



La fonction assert

1. Les fonctions du module assert sont nombreuses. En voici quelques-unes :

equal	isNull	isNotObject	isNotBoolean
notEqual	isNotNull	isArray	typeOf
strictEqual	isNaN	isNotArray	notTypeOf
notStrictEqual	isNotNaN	isString	instanceOf
isTrue	isUndefined	isNotString	notInstanceOf
isNotTrue	isFunction	isNumber	match
isFalse	isNotFunction	isNotNumber	notMatch
isNotFalse	isObject	isBoolean	etc...



Assertions - Chai

1. Mocha a servi à rédiger le protocole de test en anglais. Cependant, c'est avec Chai qu'il va être implémenté.
2. Installation : `npm install --save-dev chai`
3. Chai est une bibliothèque d'assertions, c'est-à-dire de fonctions qui effectuent des vérifications et lèvent des exceptions si celles-ci ne sont pas validées.
4. Chai fournit deux interfaces (`should` et `expect`) qui rendent le code concis et lisible.



Chai - should

1. Ajoute une propriété should à tous les objets, ce qui permet de tester grâce à de nombreuses méthodes.

```
require('chai').should();  
  
(4).should.be.less(42);
```

Chai - expect

- Comparable à la précédente mais n'ajoute pas une propriété à tous les objets. C'est pour cela que cette interface est privilégiée à la précédente.

```
var expect = require('chai').expect;  
  
expect(4).to.be.less.than(42);  
  
expect(myObject).to.not.exist;  
  
var myObject = { foo: 'bar' };  
  
expect(myObject).to.be.an('object');  
expect(myObject).to.have.property('foo').that.is.a('string');
```

Atelier : tests



A cartoon illustration of a person with dark hair, wearing a purple shirt, sitting at a light blue desk. They are facing a vintage-style computer setup consisting of a CRT monitor on top of a keyboard unit. A mouse sits on the desk to the right of the keyboard. An open book lies on the desk to the left of the computer. The person is looking at the screen, with their hands resting on the keyboard.

© m2iformation JVS-NOD

171



Notre expertise est votre avenir



Outils de développement



JVS-NOD



Automatisation des tâches : Gulpjs

1. Gulp est un lanceur de tâches conçu principalement pour la compilation de sources.
2. Installation :
 - Installation en globale : `npm install --global gulp`
 - Installation locale : `npm install --save-dev gulp`
 - Pour compresser les scripts : `npm install --save-dev gulp-uglify`
 - Pour compresser les styles : `npm install --save-dev gulp-clean-css`
 - Pour compresser les images : `npm install --save-dev gulp-imagemin`



Gulp : création d'une tâche

1. Créer un fichier gulpfile.js à la racine du site :

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

gulp.task('foo', function(done) {
    console.log('Ma tâche foo');
    done();
});
```

2. Lancer la tâche : **gulp foo**

```
C:\Users\Serge\Node\formationNJS>gulp foo
[14:34:39] Using gulpfile ~\Node\formationNJS\gulpfile.js
[14:34:39] Starting 'foo'...
Ma tâche foo
[14:34:39] Finished 'foo' after 4.07 ms
```



Exemple : compression des scripts, styles et images

```
var gulp = require('gulp');
var uglify = require('gulp-uglify');
var cleanCSS = require('gulp-clean-css');
const imagemin = require('gulp-imagemin');

gulp.task('scripts', function() {
    return
    gulp.src('public/js/dest/*').pipe(uglify()).pipe(gulp.dest('build/js/'));
});

gulp.task('styles', function() {
    return gulp.src('public/css/*').pipe(cleanCSS({force:
true})).pipe(gulp.dest('build/css/'));
});

gulp.task('images', function() {
    return gulp.src('public/img/*')
    .pipe(imagemin({optimizationLevel: 5}))
    .pipe(gulp.dest('build/img'));
});
```



Analyse du code : JSHint

1. JSHint est un linteur : il analyse le code de façon superficielle et il détecte les erreurs de syntaxe (mais aussi tout un ensemble de problèmes potentiels).
 - Installation : `npm install --global jshint`
 - Utilisation : `jshint myscript.js`

```
C:\Users\Serge\Node\formationNJS>jshint modules/clientsDb.js
modules/clientsDb.js: line 23, col 2, Missing semicolon.
modules/clientsDb.js: line 51, col 2, Missing semicolon.

2 errors
```



Redémarrage automatique : node-dev

1. **node-dev** est un petit outil extrêmement pratique qui relance un script Node quand une de ces dépendances est modifiée. A chaque fois qu'un des fichiers JavaScript composant l'application est sauvé, l'application est arrêtée puis redémarrée.
2. Installation : **npm install --global node-dev**
3. Exécution de l'application avec node-dev au lieu de Node :
 - node-dev app.js (ci-dessous, on constate que app.js a été modifiée 2 fois depuis son premier lancement).

```
C:\Users\Serge\Node\formationNJS>node-dev app.js
[INFO] 14:53:40 Restarting
[INFO] 14:54:01 Restarting
```

Atelier : outils de développement



© m2iformation JVS-NOD 178



Notre expertise est votre avenir



Débogage

Despooqaae



JVS-NOD



Introduction

1. Malgré les tests, des bogues passent toujours au travers des mailles du filet.
2. Les comprendre et les résoudre peut devenir très vite chronophage.
3. Il existe des outils adaptés pour vous aider dans cette quête, ainsi que des méthodologies pour « chasser les bogues » à grande vitesse :
 - Ajout de trace,
 - Utilisation du très puissant outil « node inspector ».

Ajout de trace

1. La première approche consiste à mettre des traces dans son code via `console.log`.
2. Pratique pour les bogues simples mais inefficace dans des cas complexes (inspection des variables ou de la pile des appels).
3. Utilisation de la bibliothèque `debug` : `npm install --save debug`
4. Dans le fichier de script (par exemple, `app.js`) :

```
var log = require('debug')('myApp');
log('***** Running *****');
```

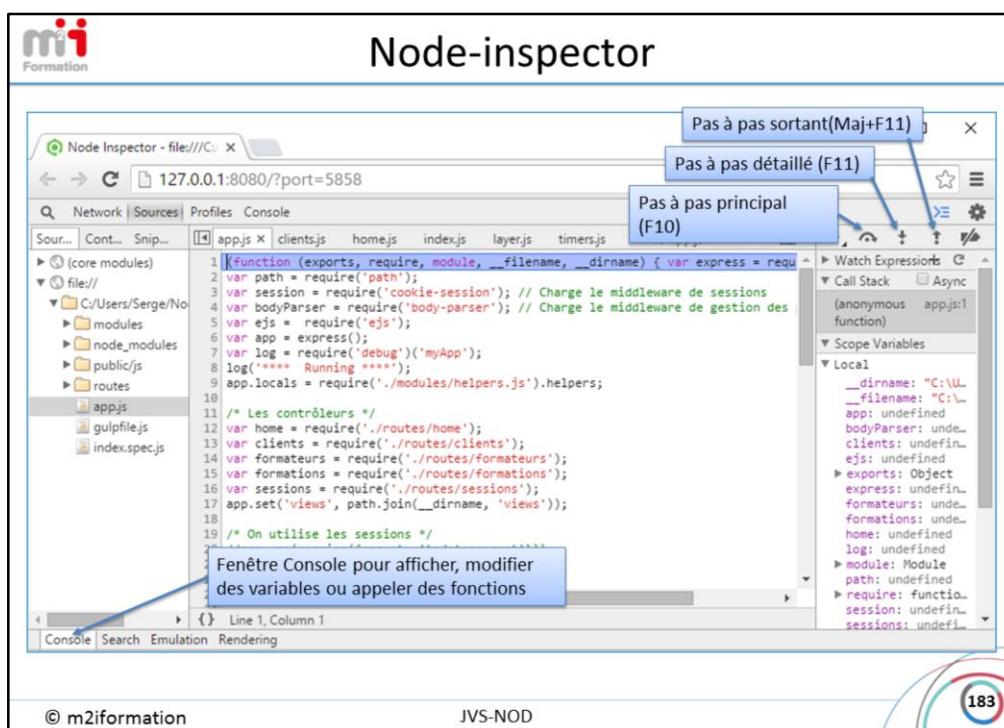
5. Lancer le script :

```
C:\Users\Serge\Node\formationNJS>SET DEBUG=myApp
C:\Users\Serge\Node\formationNJS>node app.js
myApp ***** Running ***** +0ms
```



Node-inspector

1. C'est un débogueur dédié à Node. Il communique directement avec le moteur V8 via les outils du navigateur [Chrome](#).
2. Installation : `npm install --g node-inspector`
3. Pour démarrer le débogueur : `node-debug app.js`





Node-inspector

1. Marquer un point d'arrêt :
 - Ouvrir le fichier voulu via le panneau latéral gauche et créer un point d'arrêt en cliquant sur le numéro de ligne voulue.
2. Point d'arrêt conditionnel :
 - Clic droit sur le point d'arrêt, puis « Edit breakpoint... ». Indiquer une condition d'arrêt.
3. Inspection : placer la souris au dessus d'une variable accessible via la portée courante, sa valeur s'affiche en info-bulle.
4. Intervention : dans la fenêtre console (en bas), on peut afficher puis modifier les variables. On peut même appeler une fonction.

Atelier : débogage

A cartoon illustration of a person with dark hair, wearing a purple shirt, sitting at a light blue desk. They are facing a vintage-style computer setup consisting of a CRT monitor on top of a keyboard unit, with a mouse to the right. An open book lies on the desk to the left of the computer. The person's hands are on the keyboard. The background is plain white.

© m2iformation JVS-NOD 185



Notre expertise est votre avenir



Mise en production



JVS-NOD

Introduction

1. Ce chapitre est dédié aux outils aidant à l'utilisation de Node en environnement de production, c'est-à-dire avec de nouvelles contraintes :
 - Avoir un processus lancé en permanence,
 - Superviser des applications,
 - Lancer Node au démarrage des machines,
 - Utiliser un proxy inverse...



Forever

1. Forever est un outil en ligne de commande permettant de faire fonctionner un processus Node de manière continue.
2. Installation : `npm install --global forever`
3. Utilisation : `forever start index.js`

```
C:\Users\Serge\Node\formationNJS>forever start app.js
warn:  --minUptime not set. Defaulting to: 1000ms
warn:  --spinSleepTime not set. Your script will exit if it does not stay up for at least 1000ms
info:  Forever processing file: app.js
```

Forever

1. Lister les applications Node en cours : **forever list**
2. Suivre le trace d'exécution de la commande : **forever logs *puid***
3. Redémarrer le processus Node : **forever restart *puid*** (ou **restartall**)
4. Arrêter le processus Node : **forever stop *puid*** (ou **stopall**)
5. Aide sur forever : **forever -h**

puid correspond au numéro du processus alloué par forever (et non celui alloué par Windows). Pour connaître le numéro du processus : **forever list**



Recettes - Reverse proxy

1. Le proxy inverse est un concept simple et fort utile. Au lieu qu'un client fasse des requête directement sur l'application Node, il existe un intermédiaire entre les deux. Cette façon de faire est courante parce que très puissante.
2. Tout d'abord, on peut intégrer des applications Node dans l'infrastructure existante, par exemple Apache. Ensuite, cela rend possible l'utilisation du cache au niveau de ce proxy afin d'accélérer les traitements et de réduire la charge.
3. Cela permet aussi d'avoir autant d'applications Node que l'on souhaite sur la même machine (sur différents ports) et d'avoir un seul point unique pour les consulter : le serveur Web qui écoute les ports 80 et 443.



Reverse proxy avec Apache

1. Vérifier que les modules mod_proxy et mod_proxy_http sont bien activés dans l'installation d'Apache.
2. Ajouter les entrées ProxyPreserveHost, ProxyPass et ProxyPassReverse dans la configuration VirtualHost du fichier httpd.conf d'Apache.



Le fichier httpd.conf d'Apache

```
<VirtualHost *:80>
    ServerAdmin monemail@mondomaine.com
    ServerName mondomaine.com
    # Demande de conserver l'entête HTTP Host intact.
    ProxyPreserveHost On
    <Proxy *>
        Order deny, allow
        Allow from all
    </Proxy>
    <Location>
        # Associe le chemin courant / au proxy HTTP situé
        # sur le port 9000 de la machine locale (localhost)
        ProxyPass http://localhost:9000/
        # Demande à Apache de mettre à jour les entêtes HTTP
        # Location, Content-Location et URI des réponses HTTP
        # venant du proxy afin qu'elles correspondent à l'URL courante
        ProxyPassReverse http://localhost:9000/
    </Location>
</VirtualHost>
```

Atelier : mise en production



**LA FORCE
D'UN
RÉSEAU**

m²i Formation
Notre expertise est votre avenir

N°Azur 0 810 007 689
PRIX D'UN APPEL LOCAL DEPUIS UN POSTE FIXE

Découvrez également l'ensemble des stages à votre disposition sur notre site

<http://www.m2iformation.fr>

© m2iformation JVS-NOD

194