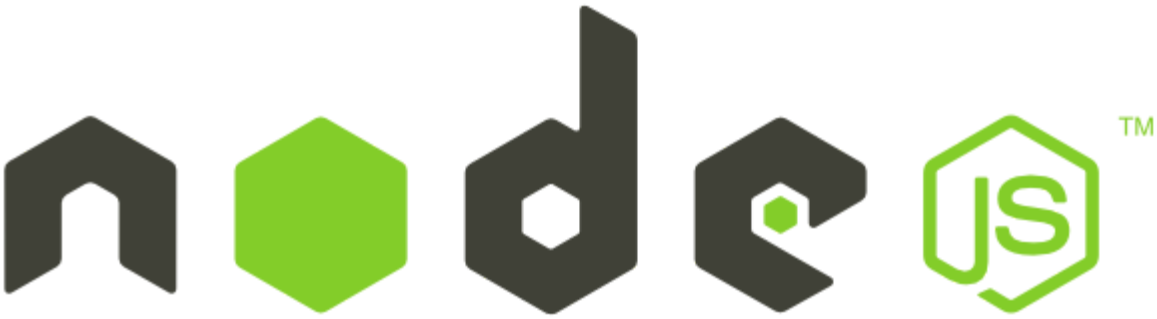
Node.js.



Programmation JavaScript côté serveur.

Cette formation vous apprendra à développer des applications réactives et performantes avec Node.js. Vous mettrez en oeuvre les concepts de programmation événementielle et asynchrone, de modularité et de routage. Vous utiliserez également les API de Node.js et assurerez la persistance de vos données dans une base NoSQL.

Objectifs pédagogiques

- Installer et configurer un serveur Node.js
- Développer des applications JavaScript côté serveur
- Mettre en oeuvre les concepts de la programmation événementielle et asynchrone
- Mettre en place un gestionnaire de routes
- Manipuler l'API de Node.js
- Gérer la persistance dans une base de données NoSQL

Présentation

Participants. (Tour de table)

Tout développeur connaissant le langage Javascript et souhaitant développer des applications performantes (haute réactivité, volumétries importantes de transactions) et orientées événements.

Prérequis.

Bonnes connaissances du langage Javascript. Une première approche d'un framework Javascript (côté client) serait un plus.

Récapitulatif matinal.

Chaque journée commence par un exercice collectif de restitution des concepts abordés la veille. L'objectif étant de renforcer (répétition) les connaissances acquises et les utiliser comme socle pour la journée à venir.

Concertation personnelle.

Votre formateur passera vous assister individuellement aussi souvent que possible.
N'hésitez pas à le solliciter pendant la journée de formation ou pour revenir sur un point particulier en fin de journée.

Votre Formateur.

Renaud Dubuis : linkedin (https://www.linkedin.com/in/renauddubuis)

Avant Propos

Ce programme est réalisé en référence à l'ouvrage **Node.js, Bonnes pratiques pour la programmation JavaScript applicative, universelle et modulaire** de **Thomas Parisot** au éditions Eyrolles.

Autres références

- (français) Programmation avec Node.js, Express.js et MongoDB : http://www.eyrolles.com/Informatique/Livre/programmation-avec-node-js-express-js-et-mongodb-9782212139945 (http://www.eyrolles.com/Informatique/Livre/programmation-avec-node-js-express-js-et-mongodb-9782212139945)
- (anglais) The Node Beginner Book : http://www.nodebeginner.org/ (http://www.nodebeginner.org/)
- (anglais) Professional Node.js: Building Javascript Based Scalable Software
- (site web) http://nodeschool.io/ (http://nodeschool.io/)
- Node Cheat Sheet (https://gist.github.com/LeCoupa/985b82968d8285987dc3) (**voir ./documentation**)

Version numérique

Pour votre confort de lecture et de manipulation ce support vous est également distribué en version numérique. (**voir ./documentation**)

Références à l’ouvrage et autres références

La formation est illustrée par la projection d’une version numérique (pdf) de votre support et l’utilisation d’autres ressources pertinentes (site internet, démonstration).

Afin de vous offrir des **repères rapides** le support mentionne les différentes références telles que:

- **cf.** **url (http://latentflip.com/loupe)** Utilisation d’une ressource externe (se reporter à l’Url mentionnée)
- **cf. pratique** Suivre, puis reproduire la manipulation (avec l’assitance du formateur)
- **cf. ressources** Se reporter au répertoire **Ressources** fourni.
- **Ex 1 : Titre exercice** Se reporter dans le répertoire **Exercices** au dossier correspondant.

NB: Lors de l’énnoncé d’un exercice la mention **CORRIGÉ** signifie qu’une correction est disponible dans le dossier correspondant, la mention ~~**CORRIGÉ**~~ signifie qu’une correction n’est pas applicable (démonstration,manipulation simple…)

Introduction

Node.js : la plate-forme JavaScript côté serveur

Node.js est une plateforme de développement ayant une histoire différente des pairs auxquels on la compare souvent : Ruby, Python, Java ou encore PHP.

Node.js (plus simplement *Node*) à initialement été créé par **Ryan Dahl** dans un but simple.

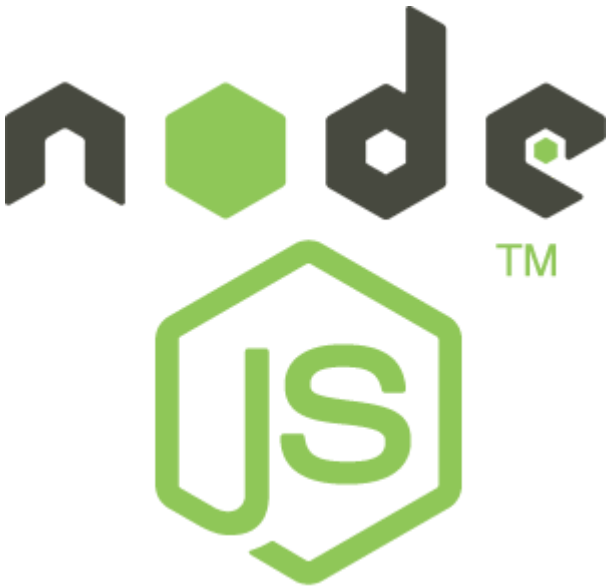
> Rendre plus simple la création de barres de chargement dans les navigateurs Web.

Le concept de pouvoir/devoir créer son propre serveur HTTP peut sembler étrange.
Le développeur est *habitué* à ce que cet aspect soit pris en charge par une solution serveur (Apache ou nginx…).

Node est devenu plus qu'une simple plate-forme de programmation : *JavaScript devient le langage de communication*.

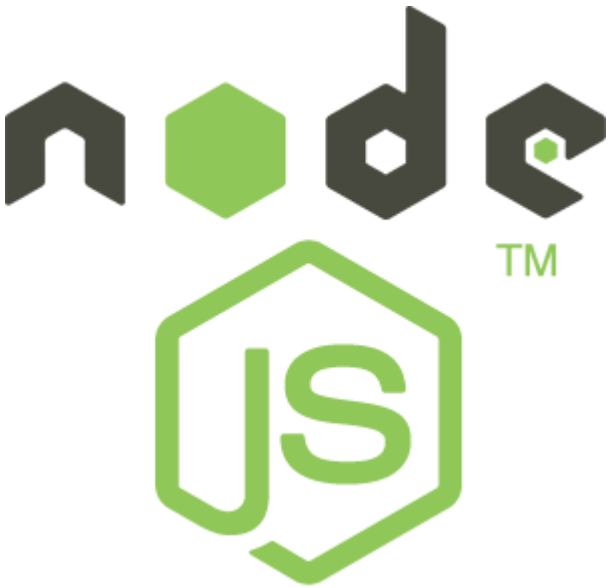
> De nombreuses entreprises ont adopté de Node : Viadeo, Paypal, LinkedIn, eBay, The New York Times, Yahoo!, Microsoft, Mozilla, Flickr ou encore Twitter.

Elles en font un usage varié : cela couvre aussi bien l'outillage, des transactions bancaires, des serveurs LDAP, des Web services ou des sites Web.



Sommaire

Table des matières.



Configuration du poste de développement.

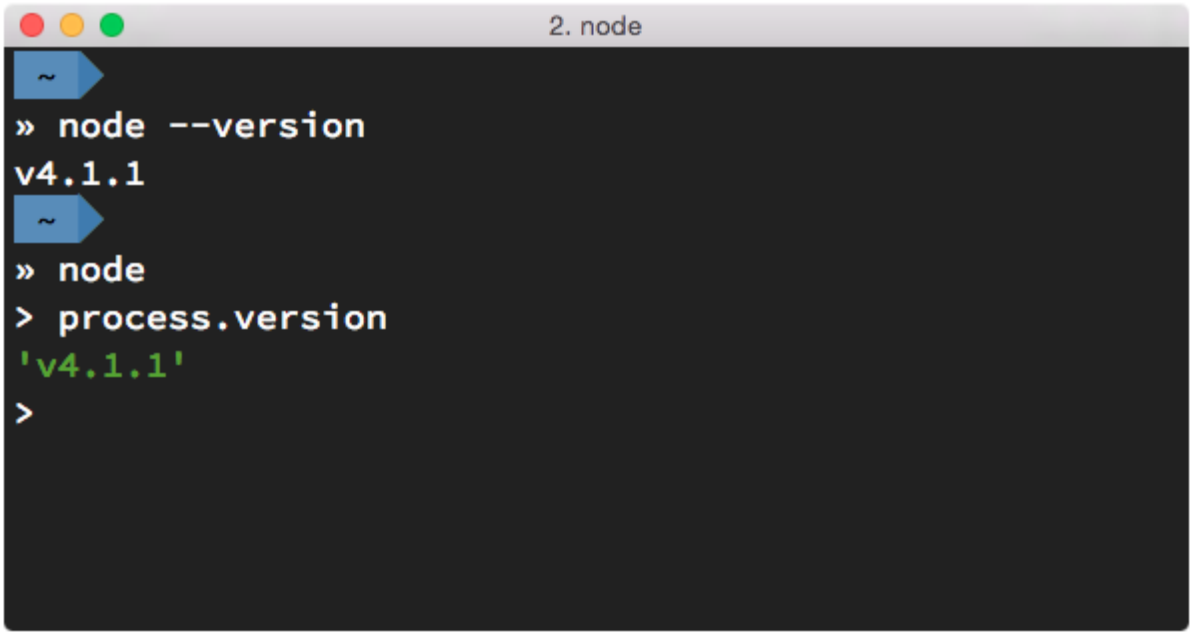
1/ Configuration du poste de développement.

1.1/ Installation de Node.JS :

Il est fortement recommandé d'utiliser un interpréteur de commandes (terminal ou shell). Les systèmes d'exploitation modernes en proposent un, y compris les versions récentes de Windows.

Si vous n'utilisez pas encore de terminal, voici une liste de recommandations non exhaustive pour vous aider :

- *OS X* : iTerm2, Terminal.app.
- *Linux* : GNOME Shell, Terminator.
- *Windows* : PowerShell, Console.



Aisance avec l’invite de commande windows.

cf. pratique

Il est utile de pouvoir ouvrir rapidement une invite de commande pointant directement sur le répertoire voulu.

Manipulation 1

```
MAJ + CLIQUE DROIT > Ouvrir une invite de commande au dossier
```

Manipulation 2

```
A l'aide de l'explorateur windows, se placer dans le dossier 'workshops'

Saisir 'cmd' + ENTRÉE dans la barre d'adresse
```

Préparer son environnement.

Installer Node n'est pas très compliqué. Il existe cependant plusieurs mécanismes d'installation.
Ces mécanismes vont du téléchargement d'un installateur à une compilation manuelle *via* un terminal.

> Les installateurs, les binaires et les sources de Node sont disponibles sur le site officiel https://nodejs.org/download/ (https://nodejs.org/download/)
> Téléchargez l'installateur adapté.

- *néophyte ou pressé(e)* : installateur du site nodejs.org, paquet fourni par le système d'exploitation ;
- *vous commencez à maîtriser* : Node Version Manager ;
- *à l'aise avec un terminal* : Node Version Manager ;
- *envie ou besoin de mettre en production* : Node Version Manager ;
- *besoin très spécifique et pointu* : compiler depuis les sources.

> Les différent modes d'installaytion sont détaillés annexe du support.

Répertoire de travail

cf. pratique

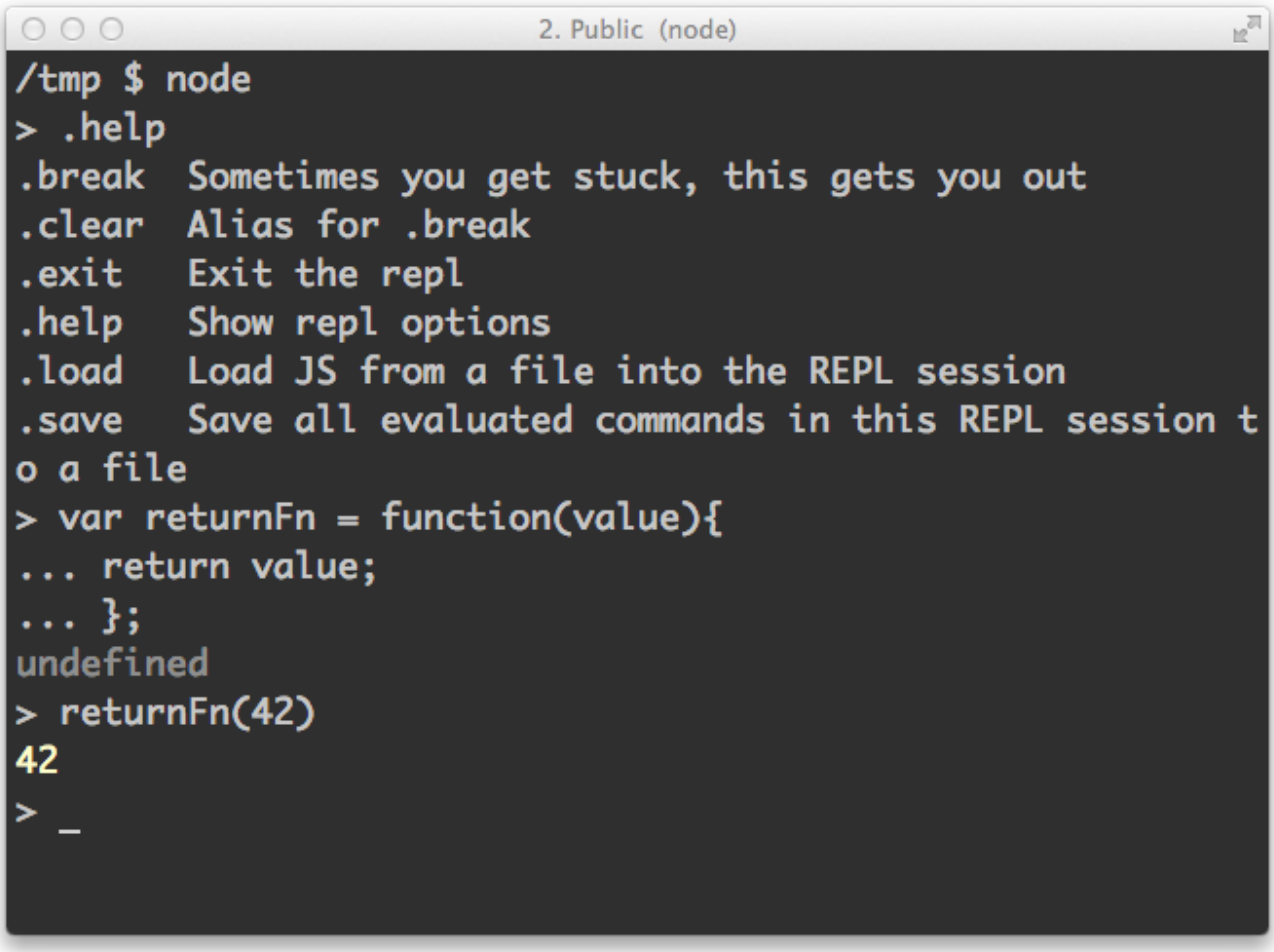
Créer un répertoire nommé **workshops** sur le bureau.

```
workshops/
│
├── concepts-es5/
│
├── concepts-es6/
│
├── node/
│
└── application/
    ├── server/
    └── client/
```

1.2/ REPL - Read Eval Print Loop

Un REPL est un interpréteur interactif exécuté dans votre terminal. Une fois invoqué, toutes les instructions écrites dans le terminal seront interprétées dès que vous presserez la touche Entrée.

L'intérêt principal d'utiliser le REPL est de prototyper rapidement du code ; code que l'on pourra éventuellement sauvegarder en tapant .save Entrée.



Le REPL lui-même est écrit en ECMAScript. Il est disponible dans le module Node natif `repl` dont la documentation est disponible à l'adresse suivante : https://nodejs.org/api/repl.html (https://nodejs.org/api/repl.html).

cf. pratique

```
Interaction avec le REPL
```

La variable globale `_` est un cas spécial.
Elle n'est prédéfinie que dans le terminal interactif (*REPL*).

Cette variable *magique* contient systématiquement la résultat de la dernière évaluation de code.

```
$ node
> 2 + 2
4
> _ + 2
6
```

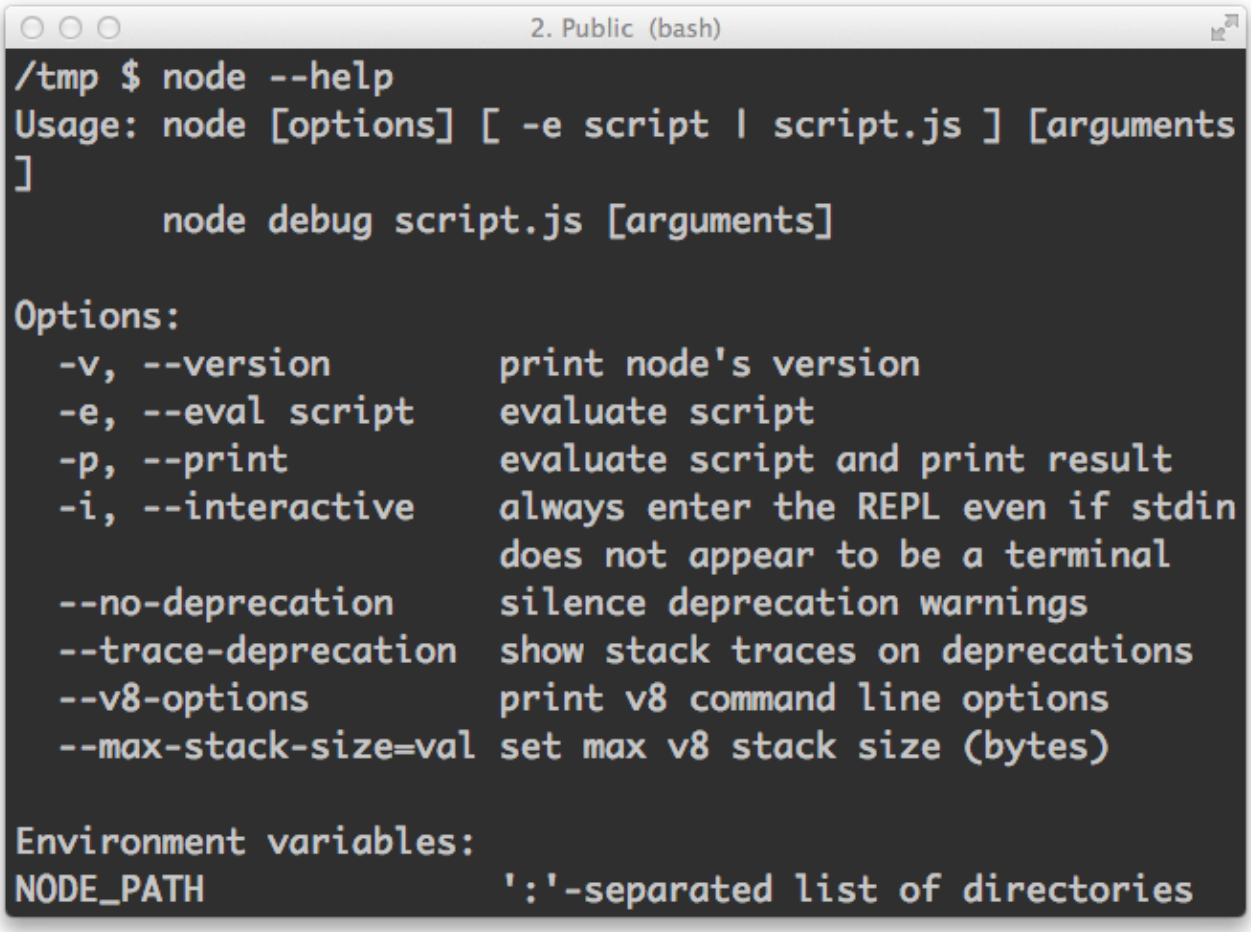
Exécution de script

L'exécution d'un script est très certainement l'invocation la plus classique de Node.

Node tente de charger et d'exécuter le fichier mentionné en argument de l'exécutable `node`. Le processus Node reste actif tant que l'*Event Loop* a des instructions à traiter dans le futur.

Un certain nombre d'options sont acceptées par l'exécutable et modifient son comportement en conséquent :

- `node debug votre-script.js` : active le mode débogage ;
- `NODE_ENV=production node votre-script` : transmets une variable d'environnement au processus, et est accessible sous la forme `process.env.NODE_ENV`.



Pour connaître l'ensemble des options disponibles de l'exécutable Node, tapez `node --help` dans votre terminal.

cf. pratique

> Exécution d'un premier programme.

Outils de développement les autres logiciels :

Programmer pour Node revient dans la majorité des cas à écrire du JavaScript. Donc même si un éditeur de texte suffit, il est intéressant de connaître l'offre en outillage autour de Node.

> Nous utiliserons Sublime Text, d'autre enviroonement de développement sont présentés en annexe.

cf. pratique

- Git **cf. ressources** attention à ajouter **git au PATH windows!**
- Node.js **cf. ressources cf. §1**
- MongoDB **cf. ressources**
- MongoDB Compass **cf. ressources**
- MongoChef **cf. ressources**
- Chrome Canary **cf. ressources**
- Sublime Text 3 **cf. ressources**

Vérifier des installation :

cf. pratique

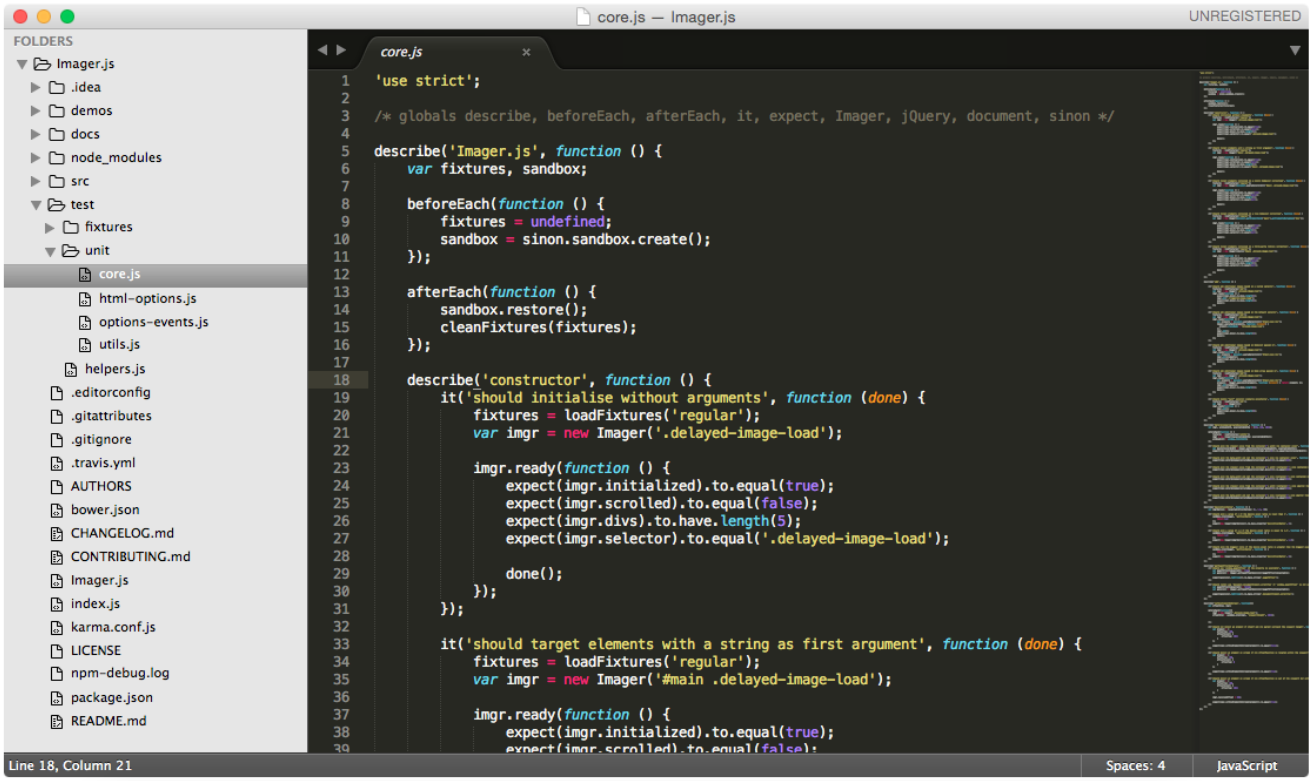
Vérifier l'installation de git, node et npm (installation par node), ouvrir une invite de commande et saisir les commandes suivantes :

```
$> node --version
x.x.x

$> npm --version
x.x.x

$> git --version
x.x.x
```

√ Installation réussie !

1.3/ Découverte de l’éditeur Sublime Text 3

Sublime Text (https://www.sublimetext.com/) (version 3 en bêta) apporte toutes les fonctionnalités d’un éditeur moderne, sans nécessiter un investissement de formation.



Fonctionalités principales:

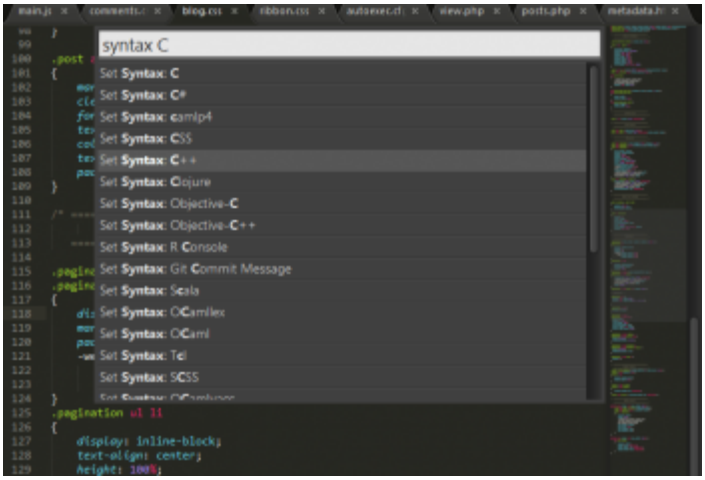
- Minimap
- Palette de commande
- Gestion des fichiers et des projets
- Coder en écran scindé
- “Snippets”
- Console
- Multiplate-forme

Raccourcis utiles :

url: (https://gist.github.com/spyl94/3963465)

- Ctrl+Maj+P: palette de commande
- Alt+Maj+(1,2,3,4,5,8,9): changer de mise en page
- Ctrl+D: sélectionner l’occurence suivante dans le fichier
- Ctrl+Maj+C: montrer la console
- Ctrl+Maj+D: dupliquer la ligne courante
- Ctrl+Maj+/: commenter / décommenter

La palette de commande



C’est le centre de contrôle de votre IDE Pour ouvrir la palette de commande, appuyez sur **Ctrl+Maj+P**, tapez le nom d’une commande, les suggestions les plus cohérentes s’affichent dans une liste, validez avec la touche ENTRÉE.

- Utilisation des commandes
- Ajout de fonctionnalités (Package Control)
- Saisie intuitive

Manipulation 1

cf. pratique

Installer gestionnaire de plugin : Package Control
cf. [url (https://packagecontrol.io/installation)]

Afficher la console : **Menu > View > Show Console**

Coller le script suivant : (depuis le PDF ou en ligne)

```
import urllib.request,os,hashlib;
h = '2915d1851351e5ee549c20394736b442' + '8bc59f460fa1548d1514676163dafc88';
pf = 'Package Control.sublime-package'; ipp = sublime.installed_packages_path();
urllib.request.install_opener( urllib.request.build_opener( urllib.request.ProxyHandle
by = urllib.request.urlopen( 'http://packagecontrol.io/' + pf.replace(' ', '%20')).rea
dh = hashlib.sha256(by).hexdigest();
print('Error validating download (got %s instead of %s), please try manual install' %
```

Appuyer sur la touche : **ENTREE**

√ Installation réussie !

Package Control

Ce plugin simplifie l'installation d'autres plugins depuis le serveur central.

- Ajout de fonctionnalités.
- Ajout de fragments de code (code snippet).

Manipulation 1

Installer le plugin HTML-CSS-JS-Prettify

```
Ouvrir la palette de commande.
Saisir 'ip' comme Install Package
ENTRÉE
Dans la nouvelle invite saisir 'HCS'
Sélectionner HTML-CSS-JS-Prettify
ENTRÉE
√ Installation réussie !
```

Manipulation 2

> Répéter autant de fois que nécessaire la manipulation 1

Installer les plugins :

- JavaScript Completion
- JavaScript Snippets
- JavaScriptNext
- JavaScript & NodeJS Snippets
- NodeJS
- AllAutocomplete
- Terminal
- SublimeREPL
- DocBlockr
- Emmet

Manipulation 3

> Découverte et utilisation des packages

Les codes Snippets

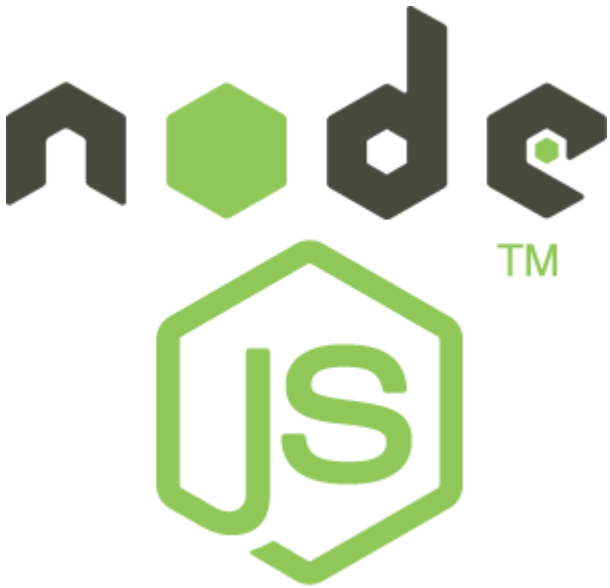
L'enrichissement de l'IDE se fait simplement par la création de snippets: **menu > tools > new snippet**.

Les code snippets sont des description XML de fragement de code à insérer au sein de fichiers textes d'extension **.sublime-snippet**

Creér un nouveau plugin et enregistrer le dans un dossier User/snippets/html

```
<snippet>
    <content>
    <!--[CDATA[
    Hello, ${1:this} is a ${2:snippet}.
    ]]-->
    </content>
    <!-- Optional: Set a tabTrigger to define how to trigger the snippet -->
    <!-- <tabTrigger>hello</tabTrigger> -->
    <!-- Optional: Set a scope to limit where the snippet will trigger -->
    <!-- <scope>source.python</scope> -->
</snippet>
```

- **content** : contenu inséré par le snippet
- marqueurs **$** : marquent l'insertion de saisie par tabulation
- **tabTrigger** : séquence de lettres déclenchant le plugin
- **scope** : restiction selon le type de document (Ctrl+Maj+Alt+P)



Rappels JavaScript

2/ Rappels (rapides) JavaScript

- **Vous**: êtes nouveau à la programmation et JavaScript? **Alors il vous faut commencer par les bases.**

> Quelque questions simples pour confirmer le socle de compétences.

- **Portée & Closure**: Saviez-vous que JavaScript utilise portée lexicale est basé sur le compilateur (pas l'interprèteur!)

  > **Pouvez-vous expliquer :**
  > L'odre de résolution des variables ?
  > Leur principe de transmition ?
  > Pourquoi les *closures* sont le résultat direct de la portée lexicale et fonctions en tant que valeurs?

- **this & Prototypes d'objets**: Pouvez-vous donner les quatre règles simples d'initialisation du mot clé `this` est lié?

- **Types & Syntaxe**: Connaissez-vous les types **natifs de JS** ? Comment vous sentez-vous avec les nuances syntaxique en JS ?

- **Async & Performance**: Comment utilisez-vous les `callbacks pour gérer votre asynchrone? Pouvez-vous expliquer ce qu'est une promesse et pourquoi / comment il résout le "callback hell"?

2.1/ Node, JavaScript et ECMAScript

> Node utilise JavaScript comme langage de développement.

Mais contrairement à Ruby, Python ou PHP dont le numéro de version annonce les fonctionnalités exploitables, pour Node , JavaScript dépend ce que la machine virtuelle *V8* est capable de comprendre.

> Chaque version de Node est associée à une version spécifique de *V8*.

De manière générale, *V8* implémente les spécifications approuvées ou en passe d'être approuvées par le comité gérant l'évolution du langage JavaScript : le *TC39*.

- https://kangax.github.io/compat-table/es5/ (https://kangax.github.io/compat-table/es5/)
- http://www.ecma-international.org/ecma-262/5.1/ (http://www.ecma-international.org/ecma-262/5.1/)

> JavaScript ou ECMAScript ? On peut lire les termes *JavaScript* et *ECMAScript* comme équivalent.
> **JavaScript et ECMAScript sont la même chose**.

JavaScript a été inventé en 1995 par Brendan Eich alors qu'il était employé de la société *Netscape Communications*.

La spécification est validée par l'organisme *Ecma International* en juin 1997 sous le nom d'*ECMAScript*, standard ECMA-262.

L'utilisation du terme *JavaScript* est resté dans le vocabulaire courant.

Standard ECMA-262 Edition 5

ECMAScript a été standardisé dans sa version 5 en décembre 2009.
Il s'agit de la version d'ECMAScript supportée depuis les débuts de Node.
La révision 5.1 de juin 2011 est une correction mineure de la spécification.

Il s'agit d'une évolution majeure, dix ans après sa précédente édition, ECMAScript 3.

ECMAScript 5 introduit le mode strict limitant fortement les effets de bord indésirables, de nouvelles fonctionnalités pour `Object` et `Array`, le support natif de *JSON* et `Function.prototype.bind`.

Standard ECMA-262 Edition 2015

La spécification *ECMAScript 2015* (*ES2015*), successivement été appelée *ECMAScript Harmony* puis *ECMAScript 6*, a été publiée en juin 2015 et succède à *ECMAScript 5*.

La page Web suivante référence l'état de l'implémentation d'ECMAScript 2015 sur différentes plates-formes, dont Node :

- https://kangax.github.io/compat-table/es6/ (https://kangax.github.io/compat-table/es6/)
- http://www.ecma-international.org/ecma-262/6.0/ (http://www.ecma-international.org/ecma-262/6.0/)

2.2/ Fondamentaux JavaScript

JavaScript est un langage riche et *dynamique* qui peut sembler déroutant comparé à d'autres langages syntaxiquement proches.
Les fondamentaux JavaScript sont les points clés de la compréhension du langage.

Scope

Le mécanisme de résolution de portée propre à javascript.

Code Hoisting

Hoisting (hissage en anglais). Phase d'optimisation du code avant son éxécution (AST) **les déclaration sont identifiées et “”hissées”**.

Object

Dans ECMAScript, tout est objet. C'est le prototype qui détermine le comportement dudit objet. Les objets peuvent être créés de manière littérale, avec la fonction Object.create ou via un constructeur.

> Les `Object` sont transmis par référence.

Primitive

Une primitive (valeur primitive ou structure de donnée primitive) est une donnée qui n'est pas un objet et n'a pas de méthode. En JavaScript, il y a 6 types de données primitives: string, number, boolean, null, undefined, symbol (nouveauté d'ECMAScript 2015).

Primitives JavaScript encapsulées dans des objets
Excepté dans les cas de null ou undefined, pour chaque valeur primitive il existe un objet équivalent qui la contient:

- *String* pour la primitive string
- *Number* pour la primitive number
- *Boolean* pour la primitive boolean
- *Symbol* pour la primitive symbol

> Les `Primitive` sont transmises par valeur.

Function

Les fonctions permettent d'isoler des portions de code (**scope**) et de les rendre réutilisables via l'utilisation de paramètres.

La déclaration d'une fonction est soumise à la portée du contexte dans lequel le bloc de code est placé. Une fonction peut être nommée `(function nomDeLaFonction(){ … }') ou anonyme (`function (){ … })`

> ES2015 introduit les "Arrow Function" ainsi que les mots clé déclaratifs `let et const`

```
'use strict';

function filterSparseArray (array) {
  return array.filter(isNotNullOrUndefined);    // <1>
}

const isNotNullOrUndefined = function (value) {
  return !isEqualTo(value, [null, undefined]);
};

function isEqualTo (value, compareWith) {
  return compareWith.some(v => v === value);  // <2>
}

(function(){                                  // <3>
  const values = [,3,,,1];
  console.log(typeof Date);                   // <4>
  console.log(isNotNullOrUndefined(null));// <5>
  console.log(filterSparseArray(values)); // <6>
})();

console.log(values);
```

1 Utilisation d'une fonction nommée comme callback de la méthode Array.map ;

2 Utilisation d'une arrow function en tant que fonction anonyme de callback ;

3 Illustration d'une **Immediatly Invocked Function Expression - IIFE** aka fonction immédiatement exécutée ;

4 Retourne 'function' ;

5 Retourne false — la valeur étant nulle ;

6 Retourne [3, 1] — un tableau filtré des valeurs nulles ou égales à undefined ;

7 Lève une exception car la constante values n'est pas définie dans ce scope.

Code hoisting

Lorsque le moteur rencontre une déclaration de variable `var = ...`, il créé un binding dans le contexte courant.

Sinon il cherche récursivement dans les contextes parents jusqu'à trouver la déclaration en question et référence le binding du contexte contenant la déclaration.

```
var foo;
(function(){
    foo = 1;    // creates a binding with the foo declared on parent context
})();
```

Hissage des déclarations

```
foo();
var bar = 'Hello World';
function foo(){
  console.log(bar);
}

// Devient
var bar;
function foo(){
  console.log(bar);
}
foo();
bar = 'Hello World';
```

Ex 2 : Code Hoisting

CORRIGÉ

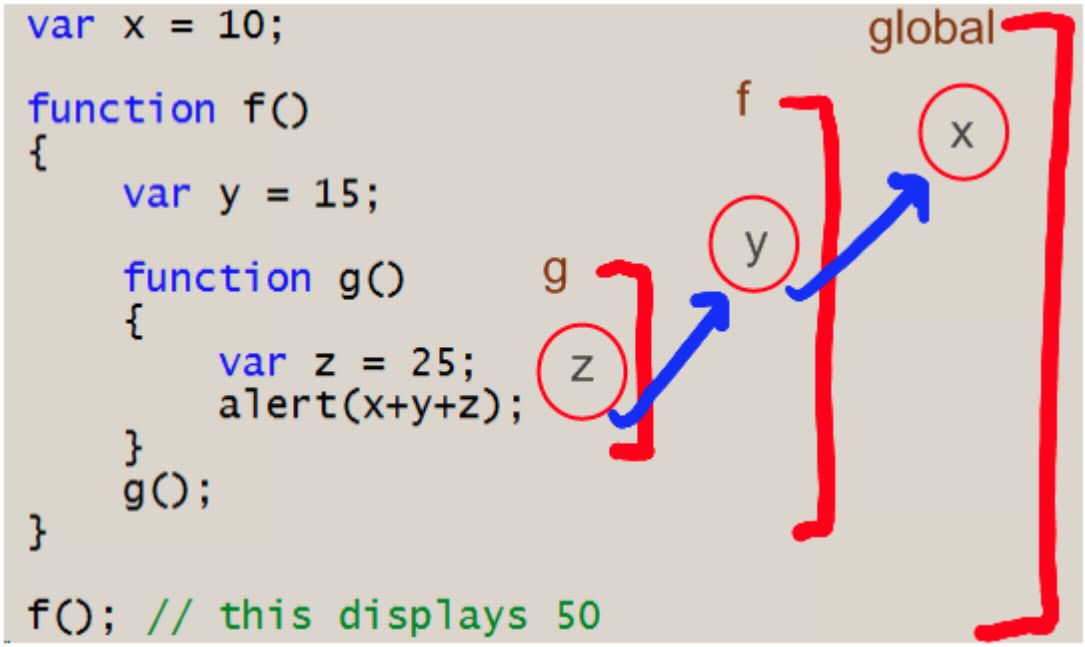
cf. [MDN Code Hoisting (https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/var)](https://developer.mozilla.org/fr/docs/Web/JavaScript/Reference/Instructions/var)

```
Ré écrire le code fourni selon les règles du Code Hoisting
```

Scope Chain et code hoisting

Le code contenu dans le bloc déclaratif d'une fonction crée une portée (ou scope) qui est invisible au contexte parent de l'exécution de cette fonction.

> Le contexte parent d'une fonction est fixé avant l'exécution (code hoisting) par la position du mot clé function.



Ex 1 : Scope Chain

```
// Prédire le résultat du script suivant.
var x = 10, y = 11, z = 12;

foo();

function foo(y){

   bar();

   function bar(){
       z = 20;
       console.log(x,y,z);
   }
}

console.log(x);
console.log(y);
console.log(z);
```

Design Pattern

NodeJS utilise le language JavaScript, nous allons revisiter les *Design Pattern JavaScript* essentielles.

JavaScript étant un language très permissif les Design Pattern utilisée répondent essentiellment à deux problématiques:

- Corriger les problématiques d'encasulation du code.
- Ré-implémenter des paradigmes (familiers) issus d'autres langages.

> **Addy Osmani (Google)** à publié un livre aux éditions **O'Reilly** et diffusé en ligne gratuitement **Learning JavaScript Design Patterns** (https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

Ex 3 : Design Pattern JavaScript

CORRIGÉ

cf. url (https://addyosmani.com/resources/essentialjsdesignpatterns/book/)

```
Produire dans autant de fichiers séparés :

- IFE
- Constructor
- Factory
- Modules Global
- Revealing Module
- Chain Function
- Object Param
- Function Memoisation
- Callback (avec setTimeout)
```

Ex 4 : Objet Global

cf. pratique

```
Nommez tous les (objets) contextes d'éxécution globaux JavaScript
auxquels vous pouvez penser et les mots clés les désignant.
```

> Rappel : l'objet **window** représente le contexte global d'éxécution du navigateur. Ce contexte peut changer selon l'implémentation des moteurs JavaScript.

ES5 “use strict” et méthodes moins connues.

> Tout vos script devraient utiliser la directive **‘use strict’;**

Le mode strict de ECMAScript 5 permet d’éxécuter un mode restrictif du moteur JavaScript.

```
// Tenter d'éxécuter ce code
"use strict";
msg = "Allo ! Je suis en mode strict !";
eval('alert(msg)');
```

Le mode strict apporte quelques changements à la sémantique « normale » de JavaScript.

> **Premièrement**, le mode strict élimine quelques erreurs silencieuses de JavaScript en les changeant en erreurs explicites.
>
> **Deuxièmement**, le mode strict corrige les erreurs limitant l’optimisation du code qui sera exécuté plus rapidement en mode strict.
>
> **Troisièmement**, le mode strict interdit les mot-clés susceptibles d’être définis dans les futures versions de ECMAScript.

ES5 méthodes moins connues.

> Les [fonctionnalités dépréciées (https://goo.gl/5ATj0X)]

Object.create

La méthode Object.create() crée un nouvel objet avec un prototype donné et des propriétés données.

```
var model = {msg:'Hello World'}
var o = Object.create(model, {
  // name est une propriété de donnée
  name: { writable: true, configurable: true, value: 'ES6' },
  // age est une propriété d'accesseur/mutateur
  age: {
    configurable: false,
    get: function() { return 10; },
    set: function(value) { console.log('Définir o.name à', value); }
  }
});
console.log(o.name,o.age,o.msg);
```

Object.defineProperty

La méthode Object.defineProperty() permet de définir une nouvelle propriété ou de modifier une propriété existante, directement sur un objet. La méthode renvoie l'objet modifié.

> Object.defineProperty(obj, prop, descripteur)
> **obj** L'objet sur lequel on souhaite définir ou modifier une propriété.
> **prop** Le nom de la propriété qu'on définit ou qu'on modifie.
> **descripteur** Le descripteur de la propriété qu'on définit ou qu'on modifie.

Les descripteurs de données et d'accesseur sont des objets.

```
var obj = {};
// en utilisant __proto__
Object.defineProperty(obj, "clé", {
  __proto__: null, // aucune propriété héritée
  value: "static"  // non énumérable
                   // non configurable
                   // non accessible en écriture
                   // par défaut
});

// en étant explicite
Object.defineProperty(obj, "clé", {
  enumerable: false,
  configurable: false,
  writable: false,
  value: "static"
});

var valeurB = 38;
Object.defineProperty(o, "b", {get : function(){ return valeurB; },
                               set : function(nouvelleValeur){ valeurB = nouvelleValeu
                               enumerable : true,
                               configurable : true});
```

Object.defineProperties

La méthode Object.defineProperties() permet de définir ou de modifier les propriétés d'un objet directement sur celui-ci. La valeur renvoyée est l'objet modifié.

```
var obj = {};
Object.defineProperties(obj, {
  "propriété1": {
    value: true,
    writable: true
  },
  "propriété2": {
    value: "Coucou",
    writable: false
  }
  // etc.
});
```

Object.getPrototypeOf

La méthode Object.getPrototypeOf() renvoie le prototype d'un objet donné.

```
var proto = {};
var obj = Object.create(proto);
Object.getPrototypeOf(obj) === proto; // true
```

Object.keys

La méthode Object.keys() renvoie un tableau des propriétés propres à un objet (qui ne sont pas héritées via la chaîne de prototypes) et qui sont énumérables.

```
var arr = ["a", "b", "c"];
console.log(Object.keys(arr)); // affichera ['0', '1', '2']

// un objet semblable à un tableau
var obj = { 0 : "a", 1 : "b", 2 : "c"};
console.log(Object.keys(obj)); // affichera ['0', '1', '2']

// un objet semblable à un tableau avec un ordre de clé aléatoire
var an_obj = { 100: "a", 2: "b", 7: "c"};
console.log(Object.keys(an_obj)); // affichera ['2', '7', '100']

// getName est une propriété non énumérable
var monObjet = Object.create({}, { getName : { value : function () { return this.name
monObjet.name = 1;

console.log(Object.keys(monObjet)); // affichera ['name']
```

Object.seal

La méthode Object.seal() scelle un objet afin d'empêcher l'ajout de nouvelles propriétés, en marquant les propriétés existantes comme non-configurables.

Les valeurs des propriétés courantes peuvent toujours être modifiées si elles sont accessibles en écriture.

```
var obj = {
    prop: function () {},
    msg: "Hello"
  };

// On peut ajouter de nouvelles propriétés
// Les propriétés existantes peuvent être changées ou retirées
obj.msg = "World";
obj.name = "Bob";
delete obj.name;

var o = Object.seal(obj);
o === obj; // true
Object.isSealed(obj); // true

obj.coincoin = "mon canard"; // la propriété n'est pas ajoutée
delete obj.msg; // la propriété n'est pas supprimée
```

Array.prototype.map

La méthode map() crée un nouveau tableau composé des images des éléments d'un tableau par une fonction donnée en argument.

Array.prototype.filter

La méthode filter() crée et retourne un nouveau tableau contenant tous les éléments du tableau d'origine pour lesquels la fonction callback retourne true.

Array.prototype.reduce

La méthode reduce() applique une fonction qui est un « accumulateur » et qui traite chaque valeur d'une liste (de la gauche vers la droite) afin de la réduire à une seule valeur.

```
var users = [{name:'Bill',age:'10'},{name:'Bob',age:'19'},{name:'Marine',age:'25'}]

users.filter(function(e,i,a){return e.age > 18 })
     .map(function(e,i,a){ var n = e; n.name = n.name.toUpperCase(); return n; })
     .reduce(function(e1, e2, i , a) {
         return e1.name.concat(e2.name)
      });
```

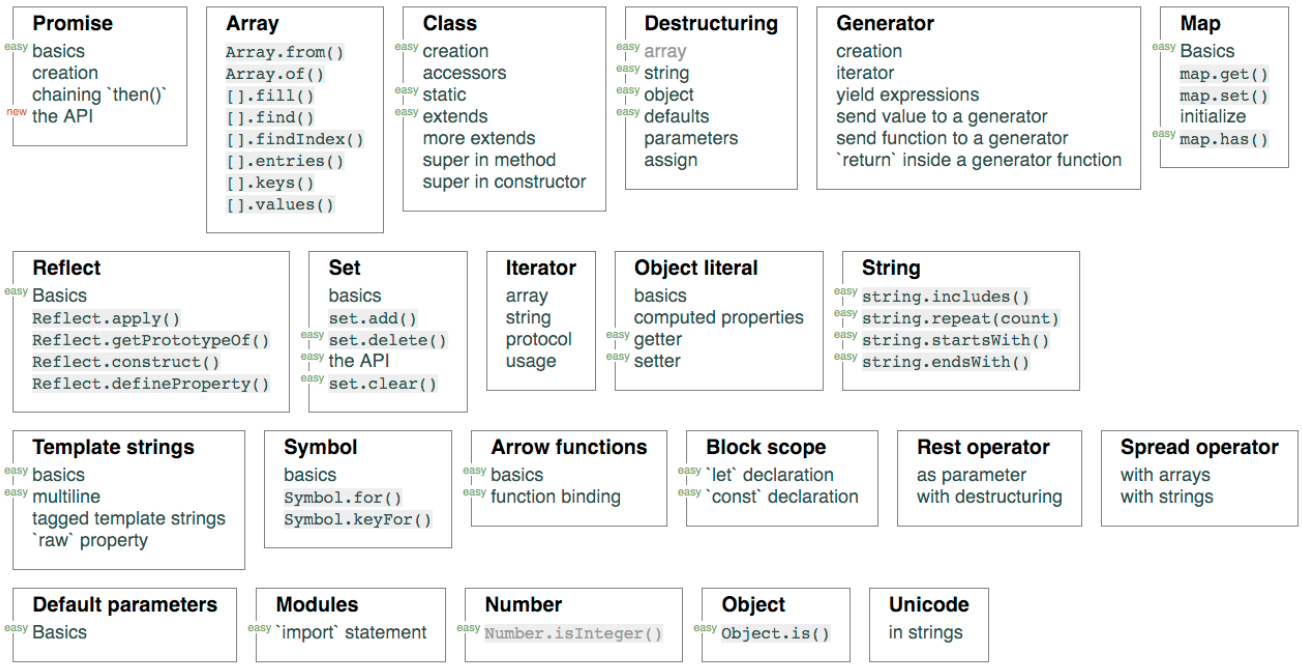
Ecma-Script 6 / 2015

NodeJS supporte partiellement la nouvelle syntaxe JavaScript ES6/2015.

La syntaxe ES6 apporte concision aux code et résoud certaines problématiques liées au scope.
le site http://node.green/ (http://node.green/) maintient un tableau détaillée du niveau de support ES6 par version de node.js.

cf. Node.js Support de ES6 (https://nodejs.org/en/docs/es6/)
cf. Table de compatibilité ES6 (https://kangax.github.io/compat-table/es6/)

Aperçu général ES6.



On notera pour le développeur node.js:

- `let/const`
- `class`
- Arrow Function Expression
- `Promise`

Variables de bloc.

L'instruction `let` permet de déclarer des variables dont la portée est limitée à celle du bloc dans lequel elles sont déclarées.

> Au niveau le plus haut (la portée globale), let crée une variable globale alors que var ajoute une propriété à l'objet.

```
var x = 'global';
let y = 'global2';
console.log(this.x); // "global"
console.log(this.y); // undefined
console.log(y);      // "global2"
```

Rappel : Le mot-clé `var` permet de définir une variable globale ou locale à une fonction (sans distinction des blocs utilisés dans la fonction).

```
for (let i = 1; i <= 5; i++) {
  setTimeout(function(){console.info(i)},1000)
}

for (var i = 1; i <= 5; i++) {
  setTimeout(function(){console.warn(i)},1000)
}
```

Redéclarer une même variable au sein d'une même portée de bloc, cela entraîne une exception `TypeError` Contrairement au corps de fonction.

```
if (x) {
  let myVar;
  let myVar; // TypeError
}

function foo() {
  let myVar;
  let myVar; // Cela fonctionne.
}
```

Faire référence à une variable dans un bloc avant la déclaration de celle-ci avec letentraînera une exception ReferenceError`. La variable est placée dans une « zone morte temporaire » entre le début du bloc et le moment où la déclaration est traitée.

> Zone morte temporaire (temporal dead zone / TDZ) concerne les erreurs liées à `let

```
function foo() {
  console.log(myVar); // ReferenceError
  let myVar = true;
}
```

Constantes

La déclaration const permet de créer une constante nommée accessible uniquement en lecture.

> **Attention:** Cela ne signifie pas que la valeur contenue est immuable, uniquement que l'identifiant ne peut pas être réaffecté.

Les constantes font partie de la portée du bloc (comme les variables définies avec let).

```
const myVar = true;
myVar = false;

const myObj = {};
myObj.property = false;
myObj = {};
```

- Il est nécessaire d'initialiser une constante lors de sa déclaration.
- Il est impossible d'avoir une constante qui partage le même nom qu'une variable ou qu'une fonction.(Au sein d'une même portée)

> **Usage:**
> On préférera `let` pour les variables et `const` pour les fonctions.

```
const foo = function foo() {
  let myVar = true;
  console.log(myVar);
};
```

Assignation destructurée.

L'affectation par décomposition (destructuring en anglais) est une expression JavaScript permettant d'extraire des données d'un tableau ou d'un objet d'après une comparaison de forme syntaxique.

> [a, b] = [1, 2]
> [a, b, …c] = [1, 2, 3, 4, 5]
> {a, b} = {a:1, b:2}

L'intérêt de l'assignation par décomposition est de pouvoir lire une structure entière en une seule instruction.

```
var myVar= ["un", "deux", "trois"];

// sans utiliser la décomposition
var un    = myVar[0];
var deux  = myVar[1];
var trois = myVar[2];

// en utilisant la décomposition
var [un, deux, trois] = myVar;
```

Échange de variables :

```
var a = 1;
var b = 3;

[a, b] = [b, a];
```

Renvoyer plusieurs valeurs:

Il était déjà possible de renvoyer un tableau/objet mais cela ajoute un nouveau degré de flexibilité.

```
function f() {
  return [1, 2];
}
var [a, b] = f();
var x = f();
console.log("A vaut " + a + " B vaut " + b, x);

var url = "http://www.orsys.fr/formation-Ecmascript-6.asp/";

var parsedURL = /^(\w+)\:\/\/([^\/]+)\/(.*)$/.exec(url);
var [, protocol, fullhost, fullpath] = parsedURL;

console.log(protocol); // enregistre "http"
```

Ignorer certaines valeurs:

```
function f() {
  return [1, 2, 3];
}

var [a, , b] = f();
console.log("A vaut " + a + " B vaut " + b);
```

Décomposition d'objet:

```
var o = {a: 111, b: true};
var {a, b} = o;

console.log(a); // 111
console.log(b); // true

// Réaasignation
var {a: newA, b: newB} = o;

console.log(newA); // 111
console.log(newB); // true
```

Propriétés calculées:

Il est possible d'utiliser des noms de propriétés calculés, comme avec les littéraux objets, avec la décomposition.

```
let key = "a";
let { [key]: myVar } = { a: "Hello World" };

console.log(myVar); // "truc"
```

Paramètres par défaut.

En JavaScript, les paramètres de fonction non renseignés valent `undefined`. EN ES6 il est possible de définir une valeur par défaut différente.

```
//ES5
unction multiplier(a, b) {
  b = typeof b !== 'undefined' ?  b : 1;

  return a*b;
}

multiplier(5); // 5
```

> Code allégé avec ES6

```
function multiplier(a, b = 1) {
  return a*b;
}

multiplier(5); // 5
```

Les paramètres déjà rencontrés dans la définition peuvent être utilisés comme paramètres par défaut dans la suite de la définition :

```
function singulierAutoPluriel(singulier, pluriel = singulier+"s", message = pluriel +
  return [singulier, pluriel, rallyingCry ];
}

function go() {
  return ":P"
}

function avecDéfaut(a, b = 5, c = b, d = go(), e = this, f = arguments, g = this.valu
  return [a,b,c,d,e,f,g];
}
```

Auparavant, pour définir une valeur par défaut pour un paramètre, il fallait tester s'il valait undefined et lui affecter une valeur choisie le cas échéant.

> L'argument par défaut est évalué à l'instant de l'appel. Un nouvel objet est créé à chaque appel de la fonction.

Paramètres "rest".

La syntaxe des paramètres du reste permet de représenter un nombre indéfini d'arguments contenus dans un tableau.

```
function multiplier(facteur, ...lesArgs) {
  return lesArgs.map(x => facteur * x);
}

var arr = multiplier(2, 1, 2, 3);
console.log(arr); // [2, 4, 6]
```

Opérateur “spread”.

L’opérateur `spread permet de développer une expression lorsque plusieurs arguments ou plusieurs éléments sont nécessaires (respectivement pour les appels de fonctions et les littéraux de tableaux).

Syntaxe

Pour l’utilisation de l’opérateur dans les appels de fonction :

```
foo(...objetIterable);
```

Pour les littéraux de tableaux :

```
[...objetIterable, 4, 5, 6]
```

Pour la décomposition :

```
[a, b, ...objetIterable] = [1, 2, 3, 4, 5];
```

Utilisation

Tout argument passé à une fonction peut être décomposé grâce à l’opérateur et l’opérateur peut être utilisé pour plusieurs arguments.

```
function foo(v, w, x, y, z) { console.log([v, w, x, y, z])}
var args = [0, 1];
foo(-1, ...args, 2, ...[3]);

//Ignorer des paramètres

foo(...[,,2,,4]);
```

Pour créer un nouveau tableau composé du premier, on peut utiliser un littéral de tableau avec la syntaxe de décomposition, cela devient plus succinct :

```
var articulations = ['épaules', 'genoux'];
var corps = ['têtes', ...articulations, 'bras', 'pieds'];
// ["têtes", "épaules", "genoux", "bras", "pieds"]
```

Pour la création d'objet :

```
var champsDate = lireChampsDate(maBaseDeDonnées);
var d = new Date(...champsDate);
```

Pour optimiser les méthodes :

```
var arr1 = [0, 1, 2];
var arr2 = [3, 4, 5];
arr1.push(...arr2);
```

Modèles de classe et héritage. Méthodes statiques.

Les classes ont été introduites dans JavaScript avec ECMAScript 6 et sont un **sucre syntaxique** de l'héritage prototypal. Le mot clé `class` fournit une syntaxe plus claire pour utiliser un modèle et gérer l'héritage.

> **Cette syntaxe n'introduit pas un nouveau modèle d'héritage dans JavaScript !**
> JavaScript est un langage à paradigme multilpe : Orienté Objet (prototypal), impératif et fonctionnel.
> **ES6 n'introduit pas de paradigme Orienté Objet basé sur les classes**

Les classes sont des *fonctions spéciales* e la même façon qu'il y a des expressions de fonctions et des déclarations de fonctions, on aura deux syntaxe :

Déclarations de classes

```
class Polygone {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }
}
```

Expressions de classes

Si on utilise un nom dans l'expression, ce nom ne sera accessible que depuis le corps de la classe.

```
// anonyme
var Polygone = class {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }
};

// nommée
var Polygone = class Polygone {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }
};
```

Remontée des déclarations (hoisting)

> Les déclarations de classes ne sont pas remontées dans le code, il est nécessaire de d'abord **déclarer la classe avant de l'utiliser**.

```
var p = new Polygone(); // ReferenceError

class Polygone {}
```

Corps d'une classe et définition des méthodes

Le corps d'une classe, partie contenue entre les accolades, définit les propriétés d'une classe comme ses méthodes ou **son constructeur**.

> Si la classe contint plusieurs occurences d'une méthode constructor, cela provoquera une exception SyntaxError.

```
class Polygone {
  constructor(hauteur, largeur) {
    this.hauteur = hauteur;
    this.largeur = largeur;
  }

  get area() {
    return this.calcArea();
  }

  calcArea() {
    return this.largeur * this.hauteur;
  }
}

const carré = new Polygone(10, 10);

console.log(carré.area);
```

> **A noter** Il n'y pas de séparateur entre les membre d'une classe.
> **get/set** Permet l'utilisation de méthodes sous la forme de propriété.

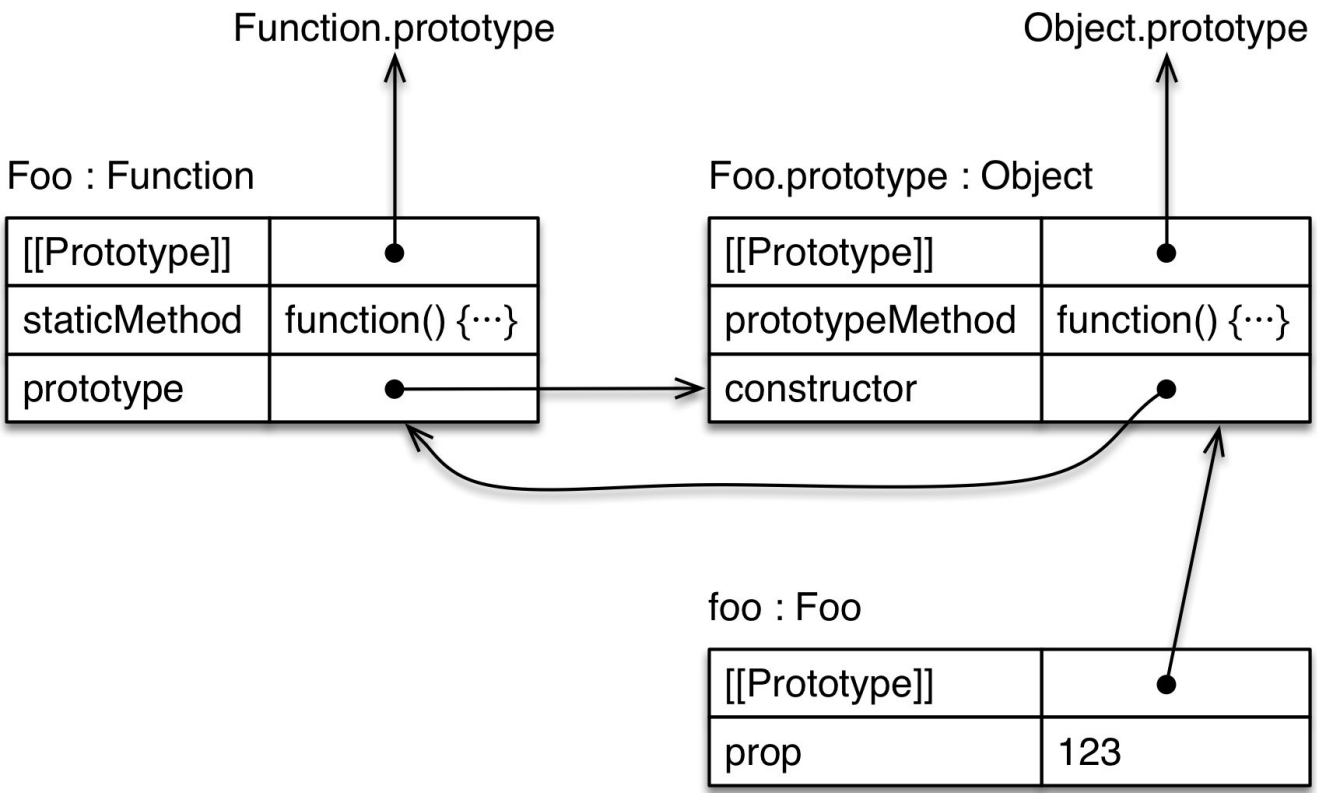
Méthodes statiques

Le mot-clé `static permet de définir une méthode statique pour une classe.

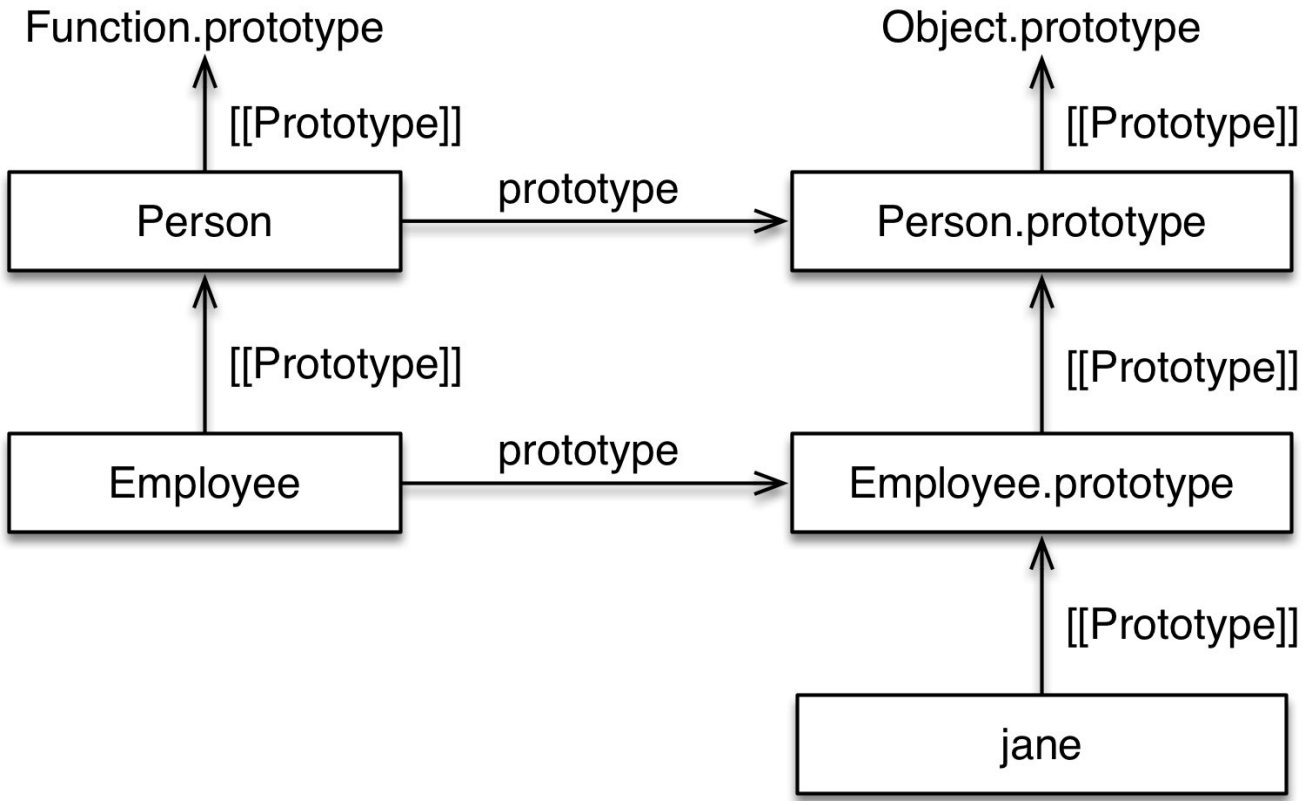
> **Les méthodes statiques** sont appelées par rapport à la classe entière et non par rapport à une instance donnée
> Ces méthodes sont généralement utilisées sous formes d'utilitaires.

```
class Point {
    constructor(x, y) {
        this.x = x;
        this.y = y;
    }

    static distance(a, b) {
        const dx = a.x - b.x;
        const dy = a.y - b.y;

        return Math.sqrt(dx*dx + dy*dy);
    }
}

const p1 = new Point(5, 5);
const p2 = new Point(10, 10);

console.log(Point.distance(p1, p2));
```

> Comprendre les méthodes statiques



Modèle prototypal



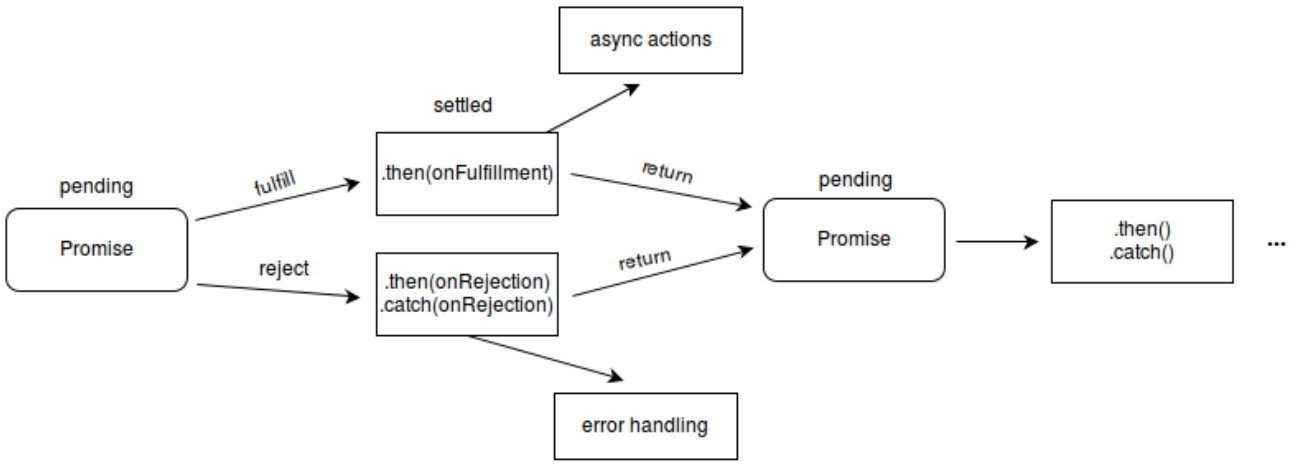

```
class Person {
    constructor(name) {
        this.name = name;
    }
    toString() {
        return `Person named ${this.name}`;
    }
    static logNames(persons) {
        for (const person of persons) {
            console.log(person.name);
        }
    }
}

class Employee extends Person {
    constructor(name, title) {
        super(name);
        this.title = title;
    }
    toString() {
        return `${super.toString()} (${this.title})`;
    }
}

const jane = new Employee('Jane', 'CTO');
console.log(jane.toString()); // Person named Jane (CTO)
```

Promise : gestion des traitements asynchrones.

L'objet Promise (pour « promesse ») est utilisé pour réaliser des opérations de façon asynchrone. Une promesse est dans un de ces états :

- en attente : état initial, la promesse n'est ni remplie, ni rompue
- tenue : l'opération a réussi
- rompue : l'opération a échoué
- acquittée : la promesse est tenue ou rompue mais elle n'est plus en attente.

```
new Promise(function(resolve, reject) { … });
```



Promise : méthodes.

Promise.all(itérable)
Renvoie une promesse qui est tenue lorsque toutes les promesses de l'argument itérables sont tenues. Si la promesse est tenue, elle est résolue avec un tableau contenant les valeurs de résolution des différentes promesses contenues dans l'itérable.

Promise.race(itérable)
Renvoie une promesse qui est tenue ou rompue dès que l'une des promesses de l'itérable est tenue ou rompue avec la valeur ou la raison correspondante.

Promise.reject(raison)
Renvoie un objet Promise qui est rompu avec la raison donnée.

Promise.resolve(valeur)
Renvoie un objet Promise qui est tenue (résolue) avec la valeur donnée.

> Gérer les appels asyncrhrones

```
const get = (url) => {
  return new Promise((resolve, reject) => {
    let xhr = new XMLHttpRequest();
    xhr.open('GET', url);
    xhr.onload = () => resolve(xhr.responseText);
    xhr.send();
  });
};

/* getTweets (Generator) */

const getTweets = (function* () {
  // 1er
  yield get('https://api.myjson.com/bins/2qjdn');
  // 2nd
  yield get('https://api.myjson.com/bins/3zjqz');
  // 3rd
  yield get('https://api.myjson.com/bins/29e3f');

})();

// Initialisation et différentes consommation

Promise.all([...getTweets]).then((valeur)=> console.log(valeur), (raison) => console.l
```

Ex 5 : Ecma-Script 6 pour Node.js

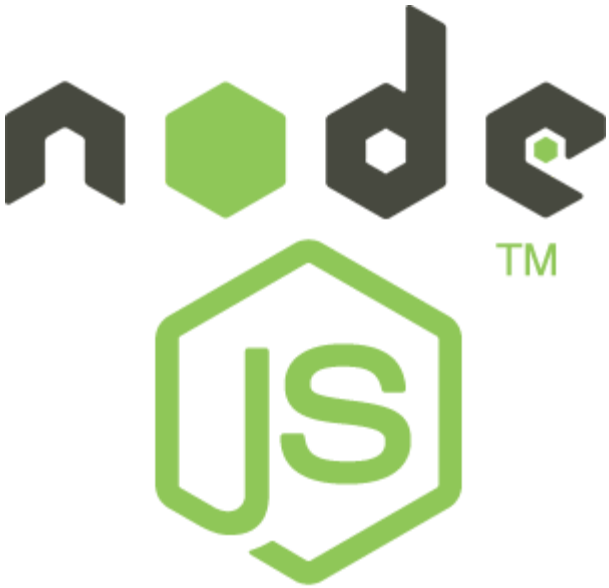
```
A l`aide de la console de Chrome Canary,
mettre en oeuvre les nouveautés suivantes
et les sauvegarder dans un fichier.

- let (strict mode only)
- const

- Classes (strict mode only)

- Map
- Set


- Promises

- Template strings

- Arrow Functions

- Object.assign
```

Ex 6 : Ecma-Script 6 pour Node.js

CORRIGÉ

cf. [url (https://nodejs.org/en/docs/es6/)](https://nodejs.org/en/docs/es6/)

```
Refactoriser le script ES5 fourni en appliquant la syntaxe ES6
```



Node.js

3/ Premiers (Seconds) pas en Node.js

3.1/ Variables globales dans Node.js

En plus des primitives ECMAScript, Node introduit des variables globales supplémentaires. Elles vous seront utiles pour faciliter le débogage ou tout simplement pour la développement et le partage de vos modules.

console

L'objet conosle `console.log` permet de tracer l'état d'une expression lors de l'exécution de son code.

Trois méthodes :

- `console.log`
- `console.error`
- `console.trace`.
- `console.dir`

`console.log` affiche une représentation textuelle d'une expression et la formate avec des motifs équivalents à la fonction C `printf()`. +
Ce contenu est envoyé vers la sortie standard, `process.stdout` :

```
console.log('ECMA%s', 'script');
```

`console.dir()` affiche les propriétés d'un objet.

`console.error` a exactement le même comportement mais redirige vers le flux d'erreur, `process.stderr`.

`console.trace` envoie l'état de la *stack trace* vers le flux d'erreur :

```
//stack-trace.js
'use strict';

function traceAtLevel(maxLevel, currentLevel){
  currentLevel = typeof currentLevel === 'number' ? currentLevel+1 : 1;

  if (currentLevel < maxLevel){
    console.log('At level', currentLevel);
    traceAtLevel(maxLevel, currentLevel);
  }
  else {
    console.trace('Level ' + currentLevel);
  }
}

traceAtLevel(5);
```

```
$ node stack-trace.js
At level 1
At level 2
At level 3
At level 4
Trace: Level 5
    at traceAtLevel (…/examples/02-first-steps/stack-trace.js:11:13)
    at traceAtLevel (…/examples/02-first-steps/stack-trace.js:8:5)
    at traceAtLevel (…/examples/02-first-steps/stack-trace.js:8:5)
    at traceAtLevel (…/examples/02-first-steps/stack-trace.js:8:5)
    at traceAtLevel (…/examples/02-first-steps/stack-trace.js:8:5)
    at Object.<anonymous> (…/examples/02-first-steps/stack-trace.js:15:1)
    at Module._compile (module.js:456:26)
    at Object.Module._extensions..js (module.js:474:10)
    at Module.load (module.js:356:32)
    at Function.Module._load (module.js:312:12)
```

Pour en savoir plus sur les méthodes et fonctions disponibles, reportez-vous à la documentation en ligne du module `console` sur [URL]#https://nodejs.org/api/console.html# (https://nodejs.org/api/console.html#).

util

Le module util contient des méthodes de mise en forme de chaînes de caractères et des méthodes utilitaires d'aide au développement.

```
var util = require("util");
var txt = util.format("Le nom est %s, le prénom est %s\n", "Bob", "Morane");
txt += util.format("Tandis que l'âge est %d", 33);
console.log(txt);
```

> Les marqueurs de format %s, %d ou %j seront remplacés par les arguments qui suivent, dans le même ordre.

La méthode `util.inspect(obj, options)` retourne une chaîne correspondant au contenu de l'objet obj.

Héritage de classes

`util.inherits(newClass, oldClass)` Permet à la classe newClass d'hériter de la classe oldClass.

Créer une classe Server qui hérite de events.EventEmitter

```
var events = require("events");
var util = require("util");
function Server () { }


util.inherits(Server, events.EventEmitter);
var server = new Server();

server.on("connexion", function() {
  console.log("Une connexion a sur le serveur");
});

server.emit("connexion");
```

Méthode Booléenne

Méthode	
util.isArray(object)	Retourne true si l'objet est de classe Array.
util.isRegExp(object)	Retourne true si l'objet est de classe RegExp.
util.isDate(object)	Retourne true si l'objet est de classe Date.
util.isError(object)	Retourne true si l'objet est de classe Error.

```
var util = require("util");
var tab = [1, 2, 3]; console.log(util.isArray(tab));
var d = new Date(); console.log(util.isDate(d));
var e = new Error(); console.log(util.isError(e));
```

Gestion des URL : module url

Méthode	
url.parse(urlStr, [parseQueryString])	Retourne un objet contenant les principaux compo- sants de l'URL transmise dans urlStr.
url.format(urlObj)	Construit une URL à partir de l'objet de paramètres.
url.resolve(from, to)	Construit une URL (relative ou absolue) selon les paramètres from et to qui lui sont transmis.

> Les deux méthodes url.parse() et url.format() utilisent un objet contenant les composants de l'URL.

Proriété	
href	URL transmise pour analyse
protocol	Protocole utilisé (par exemple, "http:").
host	Nom du host, incluant le port e(par exemple, localhost:3000).
hostname	Nom du host
port	Port utilisé (par exemple, 3000).
pathname	Partie qui suit le host (en incluant le "/"), jusqu'au "?"
search	Partie qui suit le "?" (inclus) jusqu'à la fin (par exemple, "?name=value&id=p3").
query	search, sans le "?" Toutefois, si le paramètre parseQueryString vaut true, query ne vaut plus une chaîne demais un objet.
hash	Partie qui suit l'éventuel "#" inclus (par exemple, "#label1").

Gestion des requêtes : module querystring

Le module querystring permet de manipuler la partie de l'URL correspondant à la requête (query en anglais).

Méthode	
querystring.stringify(obj, [sep], [eq])	Retourne la chaîne query correspondant à l'objet transmis, en utilisant les caractères optionnels sep comme séparateur (par défaut, "&") et eq comme signe d'égalité (par défaut, "=").
querystring.parse(str, [sep], [eq])	Opération inverse de querystring.stringify().

```
var querystring = require("querystring");
var query = querystring.stringify( {
attr1 : "value1",
attr2 : "value2"
});
console.log("Utilisation de querystring.stringify()");
console.log("La query vaut : " + query);
console.log("\n");
console.log("Utilisation de querystring.parse()");
var obj = querystring.parse(query); console.dir(obj);
```

Gestion des chemins : module path

Le module path permet de manipuler les noms de fichiers et les chemins associés par des méthodes explicites.

- path.normalize(p)
- path.join([path1],[path2], […])
- path.dirname(p)
- path.basename(p)
- path.extname(p)
- path.sep

> https://nodejs.org/api/path.html (https://nodejs.org/api/path.html)

process

L'objet `process` correspond à l'instance de l'environnement Node en cours d'exécution.

Il permet de s'interfacer avec le système, en écoutant les évènements qu'il envoie au processus ou en écoutant les évènements que Node s'apprête à envoyer au système d'exploitation.

Le tableau `process.argv` contient le chemin du script exécuté ainsi que les différents arguments transmis à Node :

```
'use strict';

console.log(process.argv);
```

Exécutons ce même script avec différents arguments :

```
node process.js <1>
node process.js argument1 "argument 2" --option1 <2>
```

<1> Retourne `[ 'node', '…/examples/chapitre-02/process.js' ]` ;
<2> Retourne `[ 'node', '…/examples/chapitre-02/process.js', 'argument1', 'argument 2', '--option1' ]`.

Il n'en faut pas davantage pour bâtir votre premier programme en ligne de commande. Pour des besoins plus avancés, il existe un certain nombre de modules *npm* pour exploiter les options et arguments.

> Pour passer des arguments à Node sans qu'ils soient interprétés par le script et inversement,i l suffit de les placer au bon endroit lorsque vous construisez l'appel à l'exécutable Node :

```
node <arguments node> chemin/vers/script.js <arguments script>
```

Ces arguments se retrouveront respectivement dans `process.execArgv` et `process.argv`.

process et flux

Le deuxième ensemble d'objets à connaître est le trio `process.stdin`, `process.stdout` et `process.stderr`.
Ce sont trois flux (*Streams*) qui permettent d'accéder respectivement à l'entrée standard, à la sortie standard et à l'erreur standard.

Ils sont directement accessibles via l'interface JavaScript de Node.

Le script suivant convertit toute chaine de caractère envoyée vers l'entrée standard en lettres majuscules :

```
'use strict';

process.stdin.on('data', function(buffer){
  process.stdout.write(buffer.toString().toUpperCase() + '\n');
});
```

```
echo "Entrée standard" | node uppercase.js
```

> L'objet `process` est un contient des attributes spécifique à l'intance.
> Il est capabale rendre capable de réagir à des évènements via la méthode `process.on()`.

Cette méthode est utilisée pour écouter les évènements système et permettre à nos programmes de réagir convenablement.

Ex 8 : Méthodes utilitaires : console,util

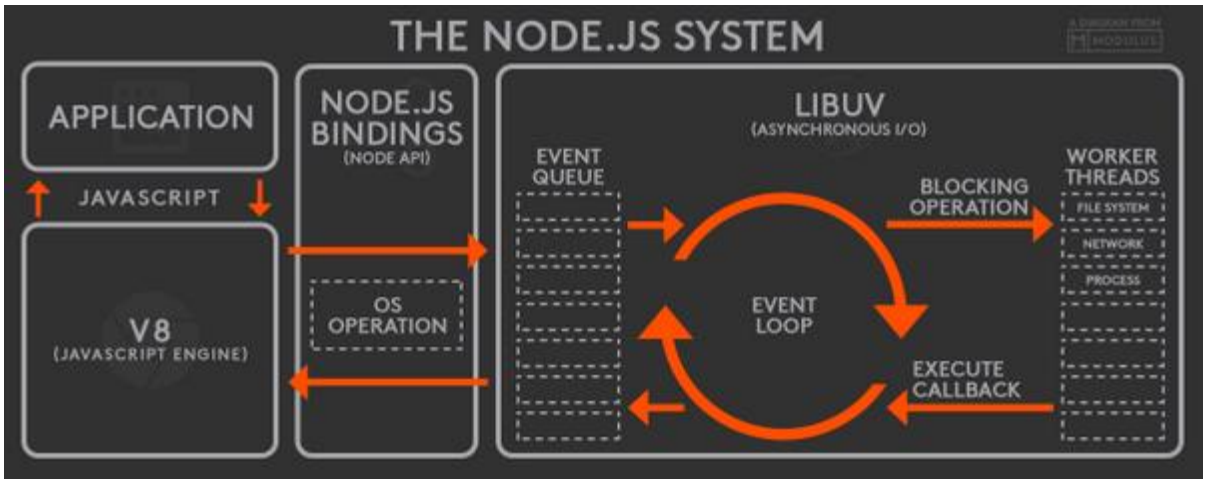
```
Découvrir et utiliser les méthodes utilitaires dans l'interpréteur.
```

__filename et __dirname

__filename et __dirname sont des constantes indiquant respectivement le chemin absolu du fichier exécuté et le chemin absolu du répertoire contenant le fichier exécuté.

setTimeout, setInterval et setImmediate

Node fournit des implémentations de `setTimeout`, de `setInterval` et de `setImmediate`. Ces fonctions de `timer` ne font pas partie de JavaScript mais de la spécification *DOM Level 0*.

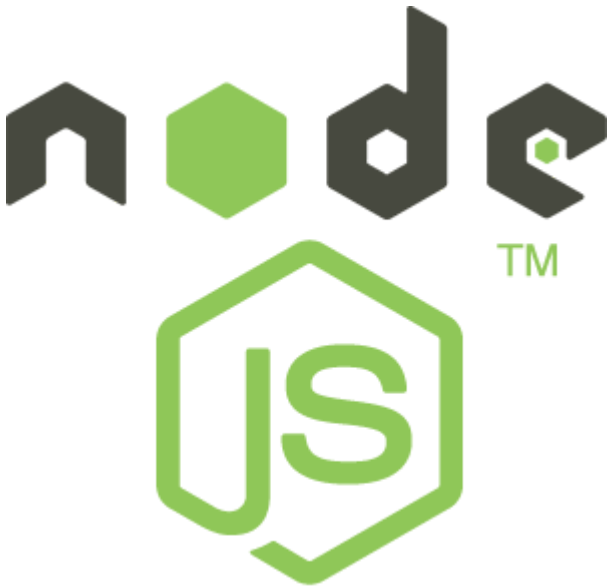


[Illustration (http://latentflip.com/loupe/)](http://latentflip.com/loupe/)

Elles programment via une API synchrone (callback queue) une fonction exécutée respectivement une seule fois dans un délai imparti, un nombre de fois indéterminé selon un interval défini si la stack est libre:

```
var print = function(message){
  return print(){
    console.log(message); <1> <2> <3>
  }
};

var timer = setInterval(print('interval'), 250);
setTimeout(print('timeout'), 200);
setImmediate(print('immediate'));
```

Pour en savoir plus sur les méthodes et fonctions disponibles, reportez-vous à la documentation en ligne du module `timers` sur [URL]#https://nodejs.org/api/timers.html# (https://nodejs.org/api/timers.html#).



Gestion des modules

4/ La gestion des modules

Les modules CommonJS sont au cœur du fonctionnement de Node.
Ils permettent d'isoler, d'empaqueter et de rendre le code réutilisable.

La spécification Module définit entre autres :

- la syntaxe de déclaration d'un module ;
- le procédé de chargement d'un module ;
- l'algorithme de résolution d'un module au sein d'un arbre de dépendance.

> http://www.commonjs.org/ (http://www.commonjs.org/)

Il s'articulent principalement autour de la fonction `require` et de l'objet `module`.

> Ils s'occupent respectivement de *charger* et de *déclarer* un module.

Techniquement parlant, un module est un fichier JavaScript dont toutes les variables sont privées et inaccessibles depuis l'extérieur.

Seules les variables exposées par `module.exports` sont publiquement accessibles.

4.1/ Propriétés et méthodes définies dans l’objet module

Propriété	Signification
id	Identifiant unique (string) associé au module. Correspond au chemin d’accès vers le module sur le serveur, ou « . » pour indiquer le module associé au main (celui qui est directement exécuté par la commande node).
exports	Objet contenant les propriétés et méthodes du module exportées vers les autres modules. Elles seront accessibles dans ces autres modules grâce à l’instruction require(modulename) qui retourne l’objet module.exports.
parent	Objet module parent de celui-ci, ou null si le module est le main.
filename	Chemin d’accès (complet) vers le module sur le serveur.
loaded	Indique si le module a été complètement chargé (true) ou non (false).
children	Tableau des objets module correspondant aux différents modules inclus dans celui-ci.
paths	Répertoires node_modules dans lesquels les modules sont recherchés. Plusieurs répertoires peuvent s’y trouver car Node les recherche d’abord dans le répertoire courant de l’application, puis remonte vers le parent, puis le parent du parent, etc.

Considérons les deux script suivant

Export : currency-format.js

```
'use strict';

var currencies = {
  FR: {
    symbol: '€',
    decimal: ','
  }
};

function formatNumber(separator, precision, number){
  return number.toFixed(precision).replace('.', separator);
}

module.exports = function setupFormatter(currencyId){
  var currency = currencies[currencyId];
  var f = formatNumber.bind(null, currency.decimal, 2);

  return function formatCurrency(amount){
    return f(amount) + currency.symbol;
  };
};
```

> Il est alternativement possible de crée des export nommés.

```
'use strict';

module.exports.value = 'Hello World';

module.exports.getValue = function (){

};
```

Consomation : currency-main.js

```
'use strict';

var setupFormatter = require('./currency-format.js');
var formatCurrency = setupFormatter('FR');

console.log(typeof currencies);
console.log(formatCurrency(12));
```

> La fonction `require` charger le fichier `currency-format.js` et assignera le résultat de l'export dans la variable de votre choix, ici, `setupFormatter`.

4.2/ Organisation des modules

L'objectif de Node est de fournir des modules natifs de bas niveau, stables et robustes. Le reste est pris en charge par la communauté. Et c'est de la communauté dont émergent les modules et les patterns utiles et efficaces.

Un module Node est **un fichier ou répertoire** contenant un fichier de description `package.json`.
Ce fichier contient notamment une propriété `main` indiquant quel fichier charger par défaut.

> Si la propriété `main` n'existe pas le système tente de trouver par défaut un fichier nommé **index.js**

Les modules Node sont par convention placés dans un répertoire `node_modules`.

Extrait du fichier `package.json` du module `lodash` :

```
{
  "name": "lodash",
  "version": "2.4.1",
  "main": "dist/lodash.js"
}
```

4.3/ Chargment des modules

La fonction `require` permet de charger des fichiers locaux, des modules JavaScript, des modules binaires ou même des fichiers JSON.

> Lorsqu'un script ext exécutée directement `require.main` prend la référence de l'objet module.

```
require.main === module
```

`require` est une fonction synchrone *et* bloquante. Cela permet de garantir l'ordre de chargement des modules.

Node optimise le chargement des modules en les mettant en cache.
Autrement dit et de manière générale, *un module est chargé une seule fois*.
L'unicité d'un module est assurée par son emplacement au sein du système de fichiers, en se basant sur la propriété `module.id`.

```
{
  "name": "lodash",
  "version": "2.4.1",
  "main": "dist/lodash.js"
}
```

> Il faut s'assurer que le code exécuté pendant le chargement d'un module soit exclusivement non bloquant pour conserver la performance applicative.

Résumé Pratique

Node est livré avec `core modules` présent au niveau de l'installation globale. Ces modules seront toujours chargé en préférence au modules locaux.

A défaut node cherche les modules dans le répertoire `node_modules` local à votre projet.

Lorsque la fonction globale `require` est invoquée, plusieurs scénarios de chargement peuvent se dérouler, en fonction de la syntaxe employée :

- un fichier local en utilisant un chemin relatif (require('./mon-fichier.js')) ;
- un module Node (require('comma-separated-values')) ;
- un module Node natif (par exemple, require('fs')).

```
var x = require('X') // depuis un module de chemain PATH
/*
1. Si X est un core module,
   a. return X
   b. STOP
2. Si X commence par './' or '/' or '../'
   a. LOAD_AS_FILE(PATH + X)
   b. LOAD_AS_DIRECTORY(PATH + X)
3. LOAD_NODE_MODULES(X, dirname(PATH))
4. THROW "not found"
*/
```

En détail : https://nodejs.org/api/modules.html#modules_file_modules (https://nodejs.org/api/modules.html#modules_file_modules)

`require.resolve()` retourn le nom du fichier qui sera chargé.

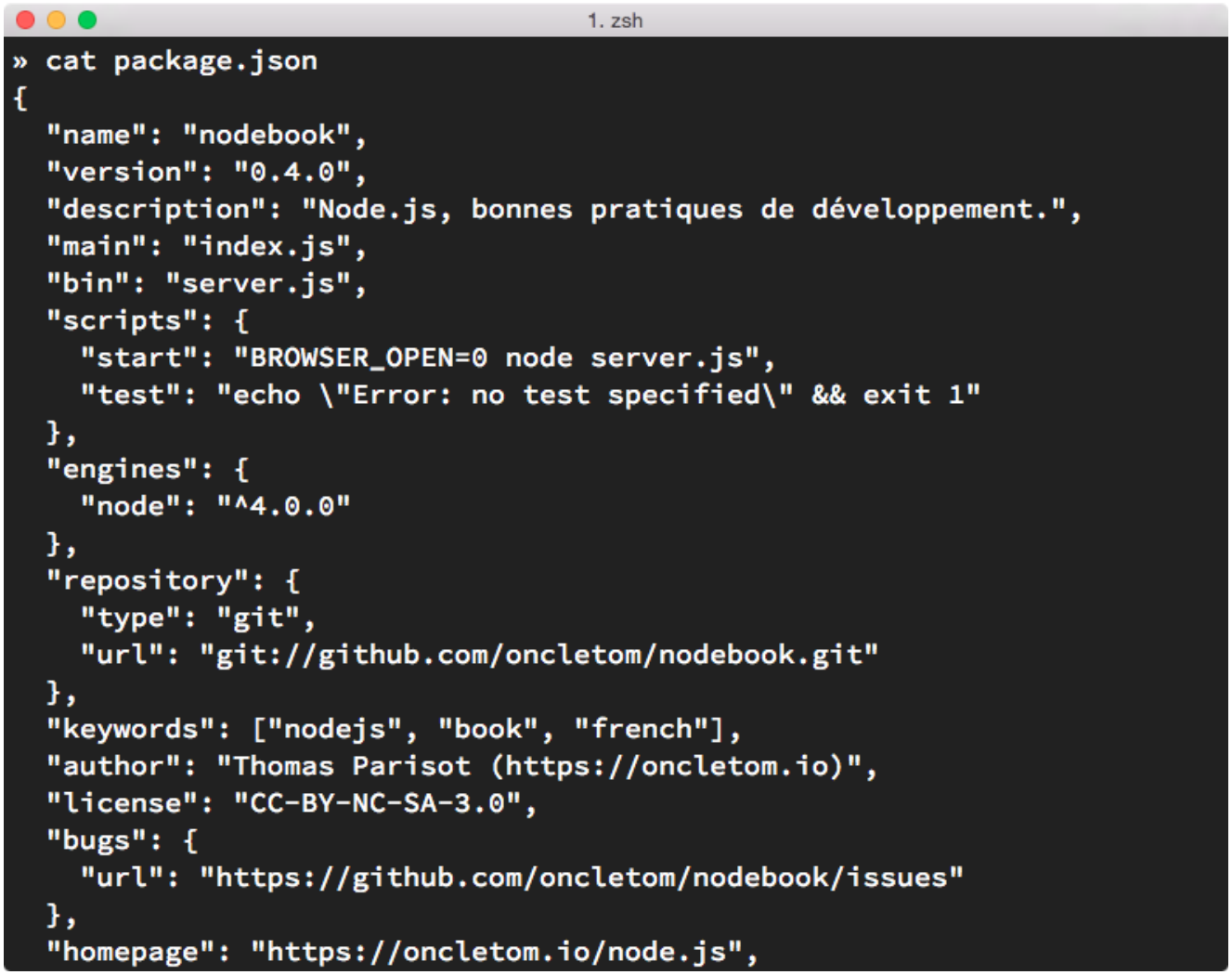
4.4/ Initialisation d’un projet

Découverte de écosystème Node.js https://www.npmjs.com/ (https://www.npmjs.com/). Les packages et leur installation globale ou locale.

- Organiser son espace de travail
- Gérer les différentes étapes du cycle de vie d’un projet Node
- Connaître et choisir des dépendances npm
- Design pattern

La pierre angulaire d’un projet Node est le fichier package.json. Incontournable, il décrit les composants essentiels du projet. Il se situe à la racine de chaque module et contient plusieurs catégories d’informations :

- *textuels* : titre, descriptions, liens et licence ;
- *version* : une chaîne respectant le fonctionnement de *node-semver* ([URL]#https://github.com/npm/node-semver# (https://github.com/npm/node-semver#)) ;
- *dépendances* : emplacement du module principal et liste explicite de modules nécessaires au bon fonctionnement du projet ;
- *actions* : commandes à exécuter lors des différentes étapes du cycle de vie du projet ;
- *divers* : données de configuration ou utilisées par des modules Node.


1. zsh
» cat package.json
{
"name": "nodebook",
"version": "0.4.0",
"description": "Node.js, bonnes pratiques de développement.",
"main": "index.js",
"bin": "server.js",
"scripts": {
"start": "BROWSER_OPEN=0 node server.js",
"test": "echo \"Error: no test specified\" && exit 1"
},
"engines": {
"node": "^4.0.0"
},
"repository": {
"type": "git",
"url": "git://github.com/oncletom/nodebook.git"
},
"keywords": ["nodejs", "book", "french"],
"author": "Thomas Parisot (https://oncletom.io)",
"license": "CC-BY-NC-SA-3.0",
"bugs": {
"url": "https://github.com/oncletom/nodebook/issues"
},
"homepage": "https://oncletom.io/node.js",

Ex 9 : Installation Globale

```
Installer globalement le module lite-server
> npm install --global -- lite-server
Afficher la liste des modules installés globalement.
```

Initialisation

L'initialisation d'un projet Node passe par la création du fichier `package.json` et ce, quelle que soit sa taille.

> **npm init**

L'utilisation de la commande `npm init` est une bonne habitude à prendre pour débuter tout projet Node.

La commande démarre une série de questions interactives.
Certaines réponses seront pré-remplies, par exemple si un dépôt Git ou un fichier README sont détectés.

À l'issue de la série de questions, le fichier `package.json` sera créé dans le répertoire courant.
Ensuite libre à vous de le compléter avec d'autres éléments optionnels de configuration.

Configuration

Que votre projet soit public ou non, il est important de renseigner les champs décrits ci-après.
Ils indiquent aux utilisateurs les intentions du projet ainsi que l'emplacement des ressources..

- *name* : il s'agit de l'identifiant du module lorsqu'il est chargé via la fonction `require()`. Ce sera également l'identifiant npm si vous publiez ce module dans un registre public ou privé. Par exemple, si la propriété *name* vaut *nodebook*, le module se chargera via `require('nodebook')` et s'installera avec la commande `npm install nodebook` ;
- *description* : une indication textuelle des objectifs et fonctionnalités du module, écrite généralement en anglais ;
- *version* : chaîne respectant la sémantique *semver* — par exemple `1.0.0`. Nous verrons un peu plus loin dans ce chapitre comment utiliser intelligemment cette valeur pour assurer des mises à jour tout en préservant la compatibilité descendante au sein des projets dépendant de ce module ;
- *main* : emplacement du fichier Node chargé par défaut lors d'un appel à `require(<name>)`. S'il n'est pas spécifié, Node tentera de charger par défaut le fichier `index.js` ;
- *repository* : objet spécifiant le type de dépôt de code ainsi que son URL. Présent essentiellement à titre informatif ;
- *preferGlobal* : booléen indiquant si ce module a davantage vocation à être installé globalement au niveau du système ou non (`false` par défaut) ;
- *bin* : emplacement du fichier. npm effectue un lien symbolique pour rendre `<name>` disponible en tant qu'exécutable système lors d'une installation globale ;
- *private* : boolean spécifiant que le module ne doit pas être publié dans un registre npm (`false` par défaut) ;
- *dependencies* : objet représentant respectivement en clé/valeur les noms/versions des modules dont le projet dépend ;
- *engines* : objet spécifiant des contraintes de compatibilité suivant la sémantique *semver* dans lesquelles le projet s'exécute sans accroc.

Dépendances

Il existe plusieurs types de dépendances, chacune ayant sa propre utilité :

- *dependencies* : dépendances utiles à un fonctionnement en production ;
- *devDependencies* : dépendances utiles uniquement dans le cadre du développement, par exemple pour exécuter des tests ou s'assurer de la qualité du code ou encore empaqueter le projet ;
- *optionalDependencies* : dépendances dont l'installation ne sera pas nécessairement satisfaite, notamment pour des raisons de compatibilité. En général votre code prévoira que le chargement de ces modules via `require()` pourra échouer en prévoyant le traitement des exceptions avec un `try {} catch ()` ;
- *peerDependencies* : module dont l'installation vous est recommandée ; pratique couramment employée dans le cas de *plugins*. +
  Par exemple, si votre projet `A` installe `gulp-webserver` en `devDependencies` et que `gulp-webserver` déclare `gulp` en `peerDependencies`, npm vous recommandera d'installer également `gulp` en tant que `devDependencies` de votre projet `A`.

npm en résumé

Commande	Signification
npm install modulename	Installe le module indiqué dans le répertoire node_modules de l'appli- cation. Ce module ne sera accessible que pour l'application dans laquelle il est installé. Pour utiliser ce module dans une autre applica- tion Node, il faudra l'installer, de la même façon, dans cette autre appli- cation, ou l'installer en global (voir l'option -g ci-dessous).
npm install modulename -g	Installe le module indiqué en global, il est alors accessible pour tou- tes les applications Node.
npm install modulename@version	Installe la version indiquée du module. Par exemple, npm install connect@2.7.3 pour installer la version 2.7.3 du module connect.
npm install	Installe dans le répertoire node_modules, les modules indiqués dans la clé dependencies du fichier package.json. Par exemple, le fichier package.json est de la forme suivante : { "dependencies": { "express": "3.2.6", "jade": "*", "stylus": "*" } } Ceci indique de charger la version 3.2.6 d'Express, avec les dernières versions de Jade et de Stylus, lorsque la commande npm install sera lancée.
npm start	Démarre l'application Node indiquée dans la clé start, elle- même incluse dans la clé scripts. Par exemple, le fichier package.json est de la forme suivante : { "scripts": { "start": "node app" } } Ceci indique d'exécuter la commande node app, lorsque la com- mande nmp start sera lancée.

Commande	Signification
npm uninstall modulename	Supprime le module indiqué, s'il a été installé en local dans node_modules.
npm update modulename	Met à jour le module indiqué avec la dernière version.
npm update	Met à jour tous les modules déjà installés, avec la dernière version.
npm outdated	Liste les modules qui sont dans une version antérieure à la dernière version disponible.
npm ls	Affiche la liste des modules installés en local dans node_modules, avec leurs dépendances.
npm ls -g	Similaire à npm ls, mais affiche les modules installés en global.
npm ll	Similaire à npm ls, mais affiche plus de détails.
npm ll modulename	Affiche les détails sur le module indiqué.
npm search name	Recherche sur Internet les modules possédant le mot name dans leur nom ou description. Plusieurs champs name peuvent être indiqués, sépa- rés par un espace. Par exemple, npm search html5 pour rechercher tous les modules ayant html5 dans leur nom ou description.
npm link modulename	Il peut parfois arriver qu'un module positionné en global soit malgré tout inaccessible par require(). Cette commande permet alors de lier le module global à un répertoire local (dans node_modules) de façon à le rendre accessible.
	sur Internet.

Structure de projet

Chaque développeur possèdera sa propre manière de ranger et d'organiser son code.

```
├── bin
├── config
├── data
├── dist
├── doc
├── lib
│   └── models
├── node_modules
├── src
│   ├── assets
│   │   ├── images
│   │   ├── js
│   │   └── less
│   ├── routes
│   └── views
├── tests
│   ├── fixtures
│   ├── functional
│   └── unit
├── package.json
└── README
```

La suggestion d'organisation ci-avant s'explique de la manière suivante :

- *bin* : fichiers exécutables depuis un shell ;
- *config* : environnements de configuration pour éviter d'écrire ces valeurs en dur dans le code source ;
- *data* : données diverses (type binaires ou CSV) nécessaires au fonctionnement de l'application ;
- *dist* : artéfacts produits après une compilation ou un résultat de *build* — souvent une bibliothèque Node prête à l'usage pour le navigateur ou une arborescence d'application prête à être déployée ;
- *doc* : fichiers de documentation relatifs à la version courante de l'application ;
- *lib* : bibliothèque et modèles utilisées par l'application. Ce code peut typiquement grossir suffisamment pour ainsi justifier qu'il soit extrait en tant que projet(s) indépendant(s) ;
- *node_modules* : modules tiers installés automatiquement via la commande npm. Autrement dit, ne créez jamais de fichiers dans ce répertoire autrement que par la commande npm ;
- *src* : code source spécifique au projet, des routes aux vues/templates en passant par les images et le code à compiler (Sass, LESS, JSX etc.) ;
- *tests* : tests unitaires, fonctionnels et *fixtures* nécessaires à leur fonctionnement ;
- *package.json* : fichier de configuration précédemment décrit dans cet ouvrage ;
- *README* : présentation, description et documentation minimaliste — mais suffisamment pour installer, faire fonctionner et contribuer au projet.

Ajout de dépendances

Le répertoire *node_modules* contient les dépendances requises par la fonction `require()` (
Le mécanisme principal d'installation est la commande `npm install`.

L'installation d'un module est par défaut *locale* au projet.
Mais elle peut également être globale au système — nous le verrons plus tard.

> Il est toutefois recommandé d'installer les modules localement, afin de limiter leur portée uniquement au projet tout en maintenant une dépendance explicite et gérable via le fichier *package.json*.

```
npm install --save async yargs
```

La commande précédente effectue plusieurs opérations :

1. requête du registre *npmjs.com* à propos des deux modules *async* et *yargs* ;
2. si les modules existent, la version compatible la plus récente est retournée (équivalent à `npm view async version` et `npm view async version` — respectivement `0.9.0` et `1.3.1`) ;
3. téléchargement et décompression des paquets dans les répertoire `node_modules/async` et `node_modules/yargs` ;
4. introspection récursive des dépendances de ces modules et si besoin est, téléchargement et décompression dans leur répertoire *node_modules* respectif (ici `node_modules/async/node_modules` et `node_modules/yargs/node_modules`) ;
5. inscription de *async* et de *yargs* dans la configuration *dependencies* de notre fichier `package.json`.

- *–save* : enregistre le module dans la clé *dependencies* ;
- *–save-exact* : idem que `--save` mais ne rajoute pas de préfixe au numéro de version (exemple : `1.3.1` au lieu de `~1.3.1`) ;
- *–save-dev* : enregistre le module dans la clé *devDependencies* ;
- *–save-optional* : enregistre le module dans la clé *optionalDependencies*.

Variables d'environnement

Certaines applications ou modules nécessitent d'en savoir plus sur le contexte d'exécution, l'emplacement de ressources ou la manière de se connecter à des serveurs distants, par exemple.

> C'est une bonne pratique en terme de flexibilité

```
'use strict';

var server = require('http').createServer();

var defaultPort = process.env.NODE_ENV === 'test' ? 3001 : 3000;

// …

server.listen(process.env.PORT || defaultPort);
```

<1> Le port sélectionné est le port 8000 ;
<2> Le port sélectionné est le port 3001 ;
<3> Le port sélectionné (par défaut) est le port 3000.

```
PORT=8000 node src/snippets/config-env.js (1)
NODE_ENV=test node src/snippets/config-env.js (2)
node src/snippets/config-env.js (3)
```

Fichier de configuration

Opter pour un fichier de configuration fait sens si de nombreux paramètres sont à fournir à l'application — ou si certains de ces paramètres impliquent des arborescences d'objets ou de tableaux ECMAScript.

Plusieurs stratégies s'offrent à vous :

- un fichier unique pour éviter les valeurs en dur dans le code ;
- un fichier par environnement d'exécution (test, préprod, production etc.) ;
- un fichier par domaine d'application (base de données, API etc.) ;
- un fichier par domaine de sécurité (backend, frontend etc.).

```
'use strict';

var NODE_ENV = process.env.NODE_ENV || 'production';

function getConfigFromFile(env) {
  return require('./config/' + (env || NODE_ENV) + '.json');
}

getConfigFromFile();
getConfigFromFile('dev');
```

Un appel à ce module chargera :

- `production.json` lors du premier appel ;
- `dev.json` lors du second appel ;
- `test.json` lors du premier appel si `NODE_ENV=test` ;
- `dev.json` lors du second appel si `NODE_ENV=test`.

Invocation npm

L'invocation npm diffère peu de l'invocation Node précédemment décrite : elle souffre en effet des mêmes défauts de résilience.

La commande npm propose des fonctionnalités additionnelles complétant et facilitant la gestion du processus applicatif. +
Une de ces fonctionnalités est `npm start` :

```
npm start
```

Cette commande ne fait rien en soit si ce n'est exécuter la commande contenue dans la section `scripts.start` de votre fichier `package.json` :

[source]
.package.json

```
{
  "name": "module-name",
  ...
  "scripts": {
    "start": "node server.js"
  }
}
```

4.5/ La boîte à outils du développeur

Les modules utilitaires pour le développeur.

nodemon est un excellent moyen d'automatiser le rechargement de votre application quand vous la développez.
Dès qu'un fichier est modifié, le module interrompt le serveur et le relance aussitôt avec les arguments originels.

Installez le module globalement et ensuite, au lieu d'exécuter `node server.js`, exécutez `nodemon server.js`.
C'est tout !

forever est un module populaire dédié à la gestion et au monitoring de process, et pas spécifiquement Node d'ailleurs. +

La commande suivante aura pour effet de démarrer et de placer le processus `server.js` — un *daemon*.

```
forever start server.js
```

supervisor est similaire à forever. `supervisor script.js`

express Le framework Express (Express.js) est un ensemble de modules Node permettant de créer facilement des applications web avec de Node. Il est basé sur le modèle MVC (Model View Controller) qui permet de donner une architecture cohérente à une application web.

Cette architecture MVC fait qu'Express est très populaire dans la communauté Node, car de plus en plus d'applications Node sont construites sur ces bases.

node-inspector Le module node-inspector s'installe au moyen de la commande `npm install -g node-inspector`.

> Commande permettant de lancer le débogueur et de mettre en attente

```
node --debug app.js
```

> Commande permettant de lancer le débogueur et de mettre un point d'arrêt sur la première instruction du programme

```
node --debug-brk app.js
```

mongo-express fournit une interface gra- phique pour administrer la base de données, ce qui est plus pratique pour voir ou modifier le contenu de celle-ci. `install -g mongo-express`

passport est une bibliothèque de gestion d'authentification.
De nombreux modules complémentaires permettent de gérer une authentification par mot de passe, par OAuth, par compte Google Apps etc.

> https://npmjs.com/passport (https://npmjs.com/passport)

restify est une alternative à *express* si votre application n'a pour seul but que d'exposer une API REST, sans rendu HTML ou nécessitant un templating particulier.

> https://npmjs.com/restify (https://npmjs.com/restify)

Installer globalement les modules suivants :

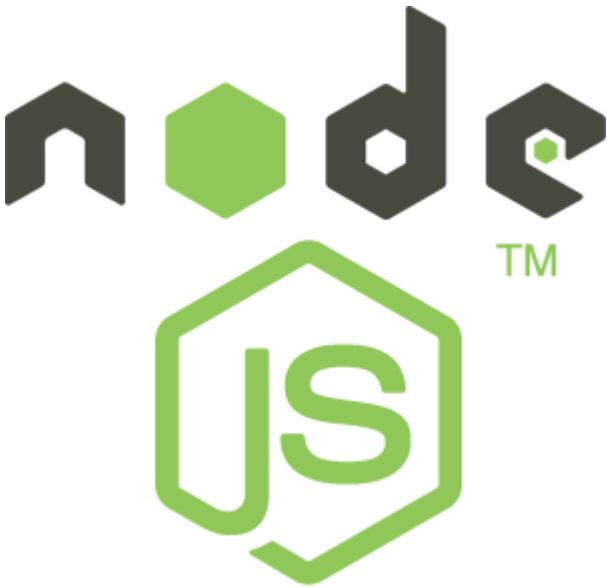
```
- connect
- express
- supervisor
- node-inspector
- mongo-express


Afficher la liste des modules installés globalement.
```

```
Lancer un script simple (console.log('Node.js')) avec supervisor.
Modifier le script
```

```
Lancer le même script simple avec node-inspector .
```

Autres modules utiles

- morgan
- body-parser
- jsonwebtoken
- winston
- nconf
- socket.io
- commander
- passport



API Node.js

5/ Tour d'horizon de l'API Node.js

5.1/ Le contexte global

cf. [global (https://nodejs.org/dist/latest-v4.x/docs/api/globals.html)]

```
Attention à l'oubli du mot clé 'var'
```

5.2/ API Modules de l'API

Si on devait la résumer en une phrase :

> Node est une API JavaScript pour manipuler des ressources système.

L'architecture se décompose en plusieurs couches, partant du plus haut niveau (exposées au développeur) et allant jusqu'au plus bas niveau (exposées au système d'exploitation) :

1. API Node
2. Interpréteur Node
3. Machine Virtuelle V8
4. libuv
5. Système d'exploitation

L'API Node correspond à des modules CommonJS écrits en JavaScript (voir ci-après) : client et serveur TCP, accès au système de fichiers, lecture de DNS, streams, buffers etc.

Ces modules natifs sont relativement bas niveau. Ils servent de base à la création d'autres modules plus faciles d'accès et partagés dans le registre npm.

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

Dans cet exemple, la fonction `require` charge l'API d'accès au système de fichier contenue dans le module `fs`.

cf. [classifiation des modules (http://www.nodewiz.biz/5-must-have-npm-node-js-modules/)]
cf. [API Stabilité (https://nodejs.org/dist/latest-v4.x/docs/api/documentation.html)]

Interpréteur Node

L'interpréteur Node est un programme écrit en C++.

L'interpréteur crée un environnement d'exécution, initialise la boucle événementielle (Event Loop, voir ci-après), lit le code JavaScript, crée l'arbre de dépendance des modules puis demande à exécuter le tout.

Machine Virtuelle V8

La machine virtuelle V8 est un compilateur JavaScript focalisé sur les performances et la sécurité. V8 a été créé par Google pour interpréter JavaScript dans son navigateur web Chrome.

Node utilise également V8 pour les mêmes raisons : parser, compiler et exécuter JavaScript. Le résultat de la compilation est retourné sous forme de fonctions et de ressources C++ manipulées par l'interpréteur Node.

libuv

Est une librairie C++ focalisée sur l'accès aux ressources système de manière non bloquante.

> **Boucle événementielle**
> La boucle événementielle (Event Loop) est un mécanisme d'exécution des tâches apporté par libuv et déléguée au système d'exploitation.

libuv implémente notamment la fameuse boucle événementielle (Event Loop, voir ci-après), la file de priorité (priority queue), délègue les accès réseaux au système d'exploitation et expose la plupart des fonctions UNIX nécessaires à la manipulation de fichiers et d'autres actions bas niveau.

> La performance de Node réside dans libuv. JavaScript n'y est pour rien, si ce n'est à travers la puissance de la machine virtuelle V8.

API Module Important

La vitesse et la croissance peuvent être des facteurs propices à l'immaturité et à l'instabilité.

> Depuis la version 4.0 de Node, des versions dites LTS (Long Term Support) sont créées tous les douze mois, contribuées pendant dix-huit mois et maintenues pendant douze mois.

Ceci garantit une plate-forme et un ensemble de fonctionnalités stables de manière prédictible, à la fois pour les projets reposant sur Node mais aussi pour l'écosystème de contributeurs de modules npm.

Node suit la convention SemVer (http://semver.org/lang/fr/) depuis la version 4.0. Précédemment, elle opérait selon le modèle de numérotation de version pair/impair :

- les versions `0.8.x`, `0.10.x`, `0.12.x` représentaient les évolutions stables de Node ;
- les versions `0.7.x`, `0.9.x`, `0.11.x` représentaient les branches de développement.

Stabilité

Node indique un indice de stabilité pour chacune de ses API publiques selon une échelle discrète graduée de 0 à 3 :

- 0 : le module est *déprécié* et ne devrait être utilisé qu'en toute connaissance de cause ;
- 1 : le module est *expérimental*, instable et nécessite des retours utilisateur ;
- 2 : le module est *stable* mais peut être sujet à des changements mineurs d'API ;
- 3 : le module est *verrouillé*, son code interne ne changera plus.

Classification [url (http://www.nodewiz.biz/5-must-have-npm-node-js-modules/)](http://www.nodewiz.biz/5-must-have-npm-node-js-modules/)

Nom	Signification
console	
debugger	Utilitaire de deboguage
Errors	Gestion des erreurs
Events	Gestion desévénements
FileSystem	Accès aux ressources
HTTP / HTTPS	Encapsulation réseau autour des protocoles
net	Gestion réseau asynchrone
path	Utilitaire de chemin système
querystring	Exploitation de querystring
Stream	Entrée/Sortie
Timers	
url	Exploitation d'URL
util	Utilitaire de développement
process	Objet d'éxécution
readline	Encapsulation de flux d'entrée

Erreurs

Les erreurs classiques JavaScript

```
<EvalError>       Evaluation du code
<SyntaxError>     Syntaxe
<RangeError>
<ReferenceError> Variables undéfinies
<TypeError>       Erreur de type
<URIError>
```

Peuvent être capturée dans des structure try/catch

```
try {
  const m = 1;
  const n = m + z;
} catch (err) {
  // Handle the error here.
}
```

Ex 14 : Pilotage du Serveur

CORRIGÉ

```
Modifier votre script pour piloter le démarage du serveur

Avec le module 'os'  récupérer l'adresse IP

Avec le module 'readline'  demander le port du serveur

Avec le module 'process'  détecter si le numéro
de port est mentionné dans les process

Avec le module 'readline'  proposer le redemarrage du serveur

Avec le module 'process'  lancer le module 'database'
sur un processus séparés
```

API Module Secondaire.

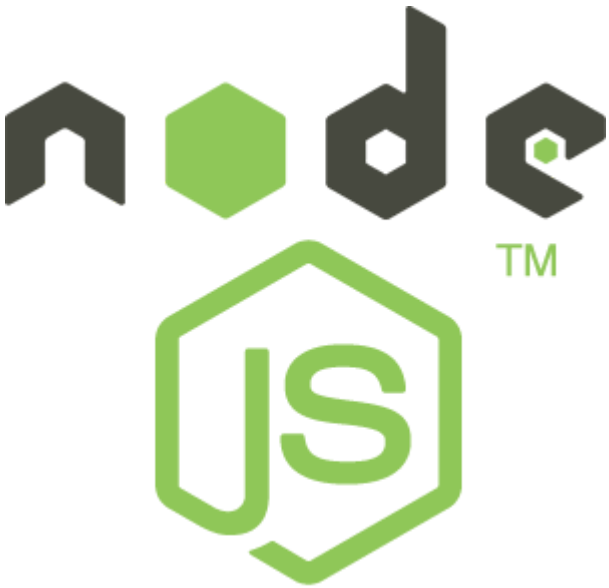
- Crypto
- DNS
- OS
- punnycode
- String Decoder
- TLS/SSL
- UDP
- V8
- VM
- zlib

API Module Ignoré.

- TTY

API Module déprécié.

- domain



Buffer

6/ Gestion des Buffer

La classe Buffer est accessible directement, sans insérer un module particulier. On crée un objet de cette classe au moyen de `new Buffer(string) ou new Buffer(size).`

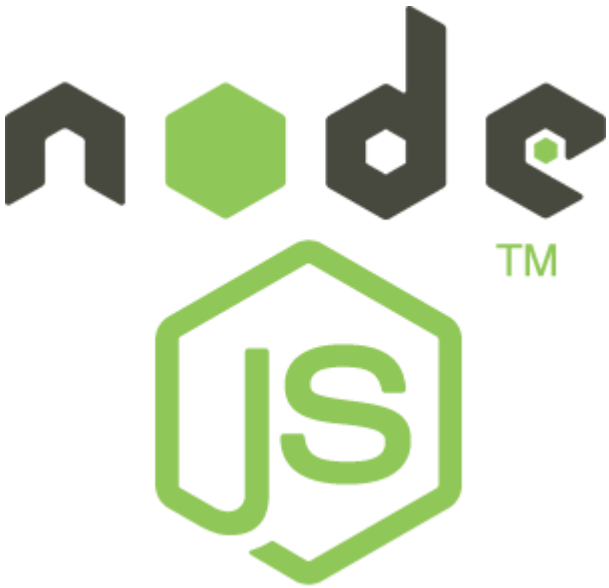
> Dans les deux cas, la taille du buffer sera fixe et correspondra à la longueur de la chaîne ou à la taille indiquée, et ne pourra pas être modifiée.

```
var buf1 = new Buffer("Bonjour");

console.log(buf1);

console.log("Taille du buffer = " + buf1.length);

console.log("\n");

var buf2 = new Buffer("Bonjour");

console.log(buf2);

console.log("Taille du buffer = " + buf2.length);
```

Une fois créé, le buffer peut être initialisé et modifié.

```
var buf = new Buffer(10);

console.log(buf);

console.log(buf.toString());

console.log("\n");

for (var i = 0; i < buf.length; i++) {

  console.log(buf[i]);

}
```

Méthode	Signification
Buffer.concat(list)	Concatène les buffers indiqués dans le tableau list et retourne un nouveau buffer résultat.
buf.write(string, offset, length, encoding)	Écrit dans le buffer buf la chaîne string , à partir de l'offset spécifié dans le buffer (par défaut, 0), et pour la longueur length indiquée pour la chaîne (par défaut, toute la chaîne). Cette dernière doit être encodée selon l'encodage utilisé ("utf8" par défaut). Les principaux encodages sont : - "utf8" (par défaut) ; - "base64" ; - "hex".
buf.copy(targetBuffer, [targetStart], [sourceStart], [sourceEnd])	Recopie une partie du buffer buf (comprise entre les indices sourceStart et sourceEnd) dans targetBuffer (à partir de l'indice targetStart). Les valeurs par défaut (optionnelles) sont les suivantes : - targetBuffer : 0. On recopie au début du buffer destination. - sourceStart : 0. On recopie à partir du début du buffer source. - sourceEnd : buf.length. On recopie la totalité du buffer source.
buf.slice([start], [end])	Retourne un nouveau buffer correspondant aux octets situés entre les indices start et end du buffer buf. Le nouveau buffer pointe en réa- lité vers l'ancien buffer buf (il n'y a pas de nouvelle allocation mémoire pour le nouveau buffer). Les valeurs par défaut (optionnelles) sont les suivantes : - start : 0. On commence à partir du début du buffer source. - end : buf.length. La fin correspond à la fin du buffer source.
Buffer.byteLength(string, [encoding])	Retourne le nombre d'octets utilisés par la chaîne string dans l'enco- dage indiqué ("utf8" par défaut).



Stream

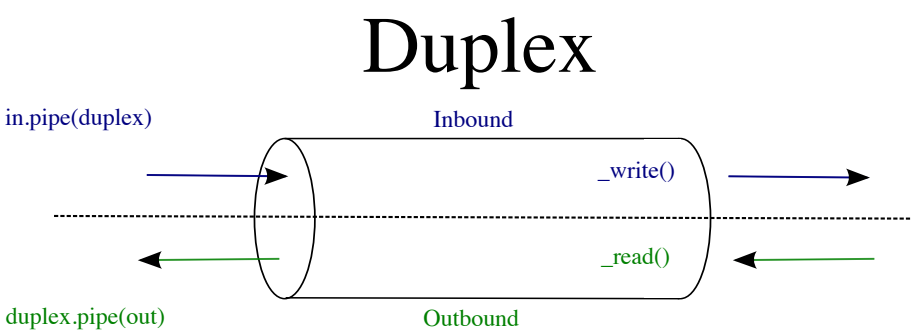
7/ Gestion des Stream

Les streams (flux en français) permettent d'échanger des informations.

Un stream est une abstraction permettant de représenter la lecture ou l'écriture de flux d'octets.

Un stream possèdela capacité d'émettre et/ou de recevoir des informations, **il hérite de classe `events.EventEmitter.**

> Par exemple, un utilisateur qui se connecte à un serveur via son navigateur HTTP, initie un stream qui permet l'échange des informations entre le client et le serveur.



7.1/ Créer un stream en lecture

Un stream est utilisable à travers le module stream.

> Pour créer un stream en lecture,on utilise la classe `stream.Readable.`

```
var stream = require("stream");

var readable = new stream.Readable();

// Indiquer  l'encodage  des chaînes de caractères en UTF-8
readable.setEncoding("utf8")

console.dir(readable);

readable._read = function(size) {

console.log("Méthode _read() appelée");

};

//Gérer la réception de données sur le stream en lecture
readable.on("data", function(chunk) {

 console.log(chunk);

});

readable.push("Bonjour1");
readable.push("Bonjour1");
```

Le paramètre chunk transmis dans l'événement data correspond au paquet d'octets reçus.

> La méthode _read() est appelée une première fois de au positionnement de l'événement data sur le stream, puis chaque fois du fait de l'envoi du paquet.

- Connaître la fin du stream en lecture avec l'événement end

Un stream en lecture peut recevoir de multiples événements data signalant la présence de données à lire sur le stream. Lorsque plus aucune donnée n'est à lire (le stream émet l'événement end, signalant ainsi la fin du stream.

7.2/ Créer un stream en écriture

On peut également créer un stream en écriture.

> Pour créer un stream en lecture,on utilise la classe `stream.Writable.`

```
var stream = require("stream");

var writable = new stream.Writable();

writable._write = function(chunk, encoding, callback) {

console.log("Appel _write");

callback();

};

console.log(writable);

//Gestion des erreurs
writable.on("error", function(error) {

console.log(error);

});

var ret1 = writable.write("1234567890", function() {

console.log("Fin write1");

});

console.log("Retour write1 = " + ret1);

//Fermeture du Stream
writable.end();

var ret2 = writable.write("2", function() {

console.log("Fin write2");

});

console.log("Retour write2 = " + ret2);
```

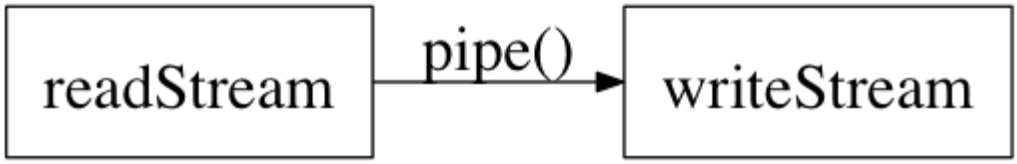
> Comme pour les streams en lecture, Node demande à ce que la méthode interne d'écriture soit définie.

_write(chunk, encoding, callback). Trois paramètres, tous obligatoires.

- Le paramètre chunk indique le buffer à écrire sur le stream.
- Le paramètre encoding précise l'encodage dans le cas où le buffer est une chaîne l'instruction write(). Cette méthode de caractères ("utf8" par défaut).
- Le paramètre callback est une fonction de callback que l'on devra appeler dans la méthode _write() pour indiquer la fin de l'écriture sur le stream.

7.3/ Connecter les Streams

http://codewinds.com/assets/article/stream-duplex.svg
(http://codewinds.com/assets/article/stream-duplex.svg)



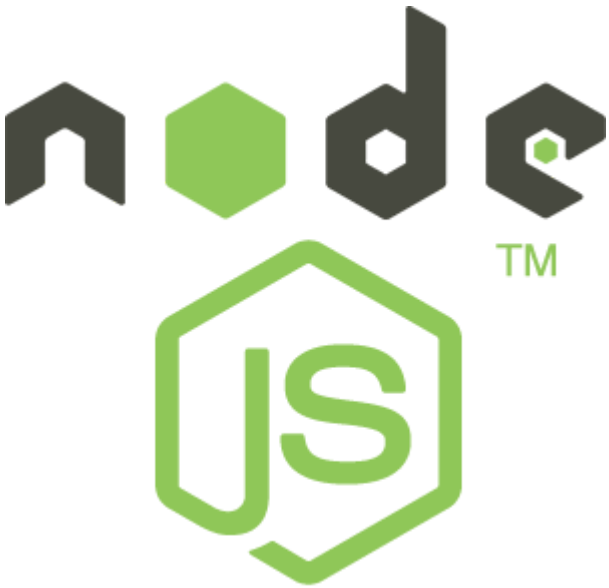
La méthode pipe() permet de connecter un stream en lecture et un stream en écriture.

```
// Rediriger les caractères écrits sur process.stdout vers un fichier logs.txt

var fs = require("fs");

var f = fs.openSync("logs.txt", "a");

process.stdout._write = function(chunk, encoding, callback) {

fs.writeSync(f, chunk);

callback();

};
```

7.4/ Duplex

Créer un stream en lecture et en écriture

```
var stream = require("stream");

var duplex = new stream.Duplex();

duplex._write = function(chunk, encoding, callback) {

console.log("Écrit sur le stream : " + chunk);

callback();

};

duplex._read = function(size) {

process.stdin.removeAllListeners("data").on("data", function(chunk) {

chunk = chunk.toString().replace(/[\d\n]/g, "");

duplex.push(chunk);

duplex.write(chunk);

});

};

duplex.on("data", function(chunk) {

console.log("Lu sur le stream : " + chunk);

});
```



Communication Evénementielle

8/ La programmation asynchrone et orientée événements

8.1/ Code bloquant et non-bloquant

Node est connu pour son API asynchrone. Toutefois le facteur le plus important est son caractère non-bloquant.

> Mais alors, *quelle différence entre code asynchrone et code non-bloquant ?*
> * *(http://latentflip.com/loupe)*)http://latentflip.com/loupe)* (http://latentflip.com/loupe)*)*

- **bloquant** : aucune instruction n'est traité, on *attend* ;
- **non-bloquant** : le reste des instructions du même cycle d'*Event loop* est traité et ainsi de suite.

cf. §3


```
Avec Node.js vous devriez  toujours favoriser une approche asynchrone !
```

8.2/ Gestion des Processus

Un programme Node consiste en l'exécution d'un seul processus, qui est la boucle de traitement des événements reçus.

Il faut décentraliser dans des processus externes les traitements trop longs grâce au module child_process.

Exécuter un processus grâce à la méthode exec()

Le module **child_process** permet de gérer les processus dans Node.

L'exécution d'un nouveau processus consiste à lancer une nouvelle commande qui s'exécutera dans le système.

Deux sortes de commandes peuvent s'exécuter :

- les commandes système, telles que dir pour afficher la liste des fichiers du répertoire courant (dir sous Windows, ls dans un système Unix) ;
- les fichiers Node écrits par nous-mêmes ou d'autres développeurs.

Méthode	Signification
child_process.exec(cmd, callback)	Exécute la commande cmd. À l'issue de l'exécution de la commande, la fonction de callback de la forme function(err, stdout, stderr) est appelée, dans laquelle : - le paramètre err indique une erreur éventuelle (null sinon) ; - le paramètre stdout représente une chaîne de caractères contenant la réponse lorsque la commande s'est bien exécutée ; - le paramètre stderr représente une chaîne de caractères contenant le message d'erreur lorsque la commande ne s'est pas exécutée correctement.

```
//Lister les fichiers d'extension .js dans un second processus

var child_process = require("child_process");

child_process.exec("dir *.js", function(err, stdout, stderr){

if (err) console.log(stderr);

else console.log(stdout);

});
```

Exécuter un fichier Node en tant que nouveau processus

```
var child_process = require("child_process");

console.log("Début du processus principal");

child_process.exec("node test2.js", function(err, stdout, stderr){

if (err) console.log(stderr);

else console.log(stdout);

});

console.log("Fin du processus principal");
```

Exécuter un processus grâce à la méthode spawn()

La méthode child_process.spawn() est similaire à la méthode exec() mais offre plus de souplesse.

> La méthode spawn() permet des interactions entre le processus père et le processus fils, telles que l'échange d'informations entre eux ou l'arrêt du processus fils à la demande du père.

Méthode	Signification
child_process.spawn("node", args)	Exécute la commande node filename afin d'exécuter le fichier Node en tant que processus fils. Le fichier à exécuter est indiqué dans le tableau args. Un objet child de classe ProcessChild est retourné par la méthode, permettant d'accéder aux streams en lecture child.stdout et child.stderr.

L'objet child retourné par la méthode spawn() permet d'accéder aux deux streams child.stdout et child.stderr qui contiendront les résultats du processus fils :

- le stream child.stdout contiendra la réponse du processus fils en cas de réussite ;
- le stream child.stderr contiendra la réponse du processus fils en cas d'erreur.

> Communication entre le processus père et le processus fils : node.js propose un API de communication basée sur les méthodes send et on.

```
//Fichier test.js (processus père)

var child_process = require("child_process");

var child = child_process.spawn("node", ["test2.js"]);

// envoi au fils de la chaîne "Orsys"

child.stdin.write("Orsys");

// récupération des éléments reçus du fils

child.stdout.on("data", function(data) {

console.log(data.toString());

});

// traitement des erreurs (obligatoire)

child.stdout.on("error", function(data) {

console.log(data.toString());

});
```

```
//Fichier test2.js (processus fils) attente des données envoyées par le père

process.stdin.on("data", function(data) {

var nom = data.toString();

// insertion de la chaîne résultat dans le stream en écriture, sur lequel le père est

process.stdout.write("Bonjour " + nom);

});
```

8.3/ Gestion des événements

La gestion des événements s'effectue au moyen du module standard **events.**

> La classe nommée EventEmitter permet de créer des objets JavaScript pouvant émettre et recevoir des événements.

Node définit déjà en standard des objets ou des classes supportant la gestion des événements. Par exemple, la classe http.Server définie dans le module http.

```
//Créer un serveur HTTP qui écoute le port 3000

var http = require("http");

var server = http.createServer(function(request, response) {

response.setHeader("Content-Type", "text/html");

response.write("<h3>Bonjour</h3>");

response.end();

});

console.log(server instanceof events.EventEmitter);
```

Créer un objet de la classe events.EventEmitter

```
var events = require("events");

var obj1 = new events.EventEmitter();

//ou

var obj1 = new (require("events")).EventEmitter();

console.log(obj1);

obj1.addListener("event1", function() {

console.log("1 - L'objet obj1 a reçu un événement event1");

});

obj1.addListener("event1", function() {

console.log("2 - L'objet obj1 a reçu un événement event1");

});

obj1.emit("event1");
```

Méthodes addListener() et emit()

- La méthode `addListener() permet d'ajouter un gestionnaire d'événements sur l'objet.
- La méthode `emit()` envoie un événement sur cet objet.

> Il est possible de supprimer un listener avec la méthode
> **obj1.removeListener("eventName", handler);**

Méthode	Signification
obj.addListener(event, listener)	Ajouter, sur l'objet qui utilise la méthode (ici, obj), un gestionnaire d'événements pour l'événement event. Lorsque cet événement se pro- duit, la fonction de callback listener se déclenche.
obj.on(event, listener)	Équivaut à addListener(event, listener).
obj.emit(event)	Déclenche l'événement event sur l'objet qui utilise la méthode (ici, obj).

> Pour supprimer tous les gestionnaires pour tous les événements inscrits depuis le lancement de l'applica- tion, il suffit d'appeler la méthode removeAllListeners() sans indiquer d'arguments.

Méthodes définies dans la classe events.EventEmitter

Méthode	Signification
obj.addListener(event, listener)	Ajoute un gestionnaire d'événements pour l'événement event (chaîne de caractères). Le paramètre listener est une fonction de callback pouvant avoir 0 à n paramètres, selon que le déclenchement de l'événe- ment par emit(event) transmet des arguments ou pas (voir la méthode emit() ci-dessous).
obj.on(event, listener)	Équivaut à addListener(event, listener).
	supprime dès que le premier événement a été déclenché. Cela permet de déclencher la fonction de traitement indiquée dans le paramètre listener une fois seulement.
obj.removeListener(event, listener)	Supprime le gestionnaire d'événements géré par la fonction représentée par listener pour l'événement indiqué.
obj.removeAllListeners(event)	Supprime tous les gestionnaires d'événements mis en place pour l'évé- nement event. Si event n'est pas indiqué, supprime tous les gestion- naires de tous les événements.
obj.emit(event, arg1, arg2, ...)	Émet l'événement event indiqué, en transmettant les éventuels argu- ments précisés dans la suite des arguments. Ces arguments pourront être récupérés dans la fonction de traitement de l'événement.
obj.setMaxListeners(n)	Indique un nombre maximal de gestionnaires d'événements pouvant être positionnés sur un objet particulier. Par défaut, on peut en position- ner au maximum 10 sur un même objet. Indiquer 0 pour ne plus avoir de limite sur cet objet.
obj.listeners(event)	Retourne un tableau contenant les références vers les gestionnaires d'événements associés à event et positionnés pour cet objet (celui qui utilise la méthode listeners()).

Ex 11 : Servuer HTTP

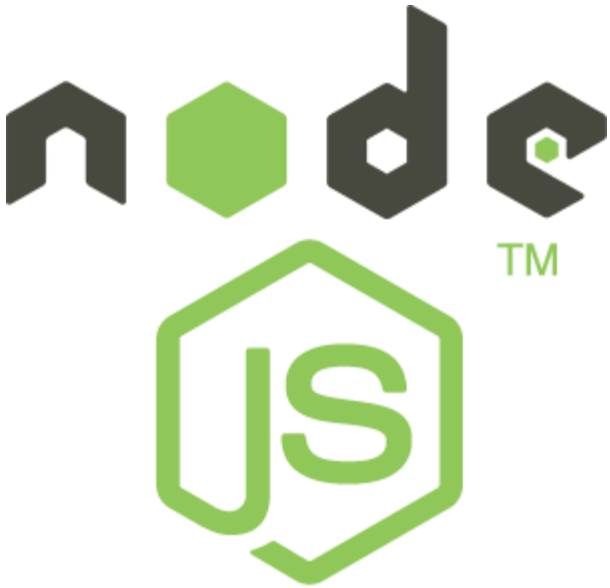
CORRIGÉ

```
Créer un simple server http -> affichant 'hello world au navigateur'

Enrichir 'liste.js' pour exporter les méthodes d'une gestion de
collection sauvegardée en mémoire:

- create
- read
- delete
- update

Initialiser vos fichers 'package.json'

$ module folder> npm init
```

Ex 12 : Communication Evénementielle

CORRIGÉ

```
Définir et propager un événements pour chaque méthode du module list
Implémenter la communication Evénementielle avec le module principale
```



Manipulation de fichiers

9/ Manipulation de fichiers

Les fichiers sont des streams particuliers, auxquels on peut adjoindre une API spécialisée définie dans
le module fs de Node (fs pour FileSystem).

Gestion synchrone
Le résultat de l'opération (une lecture, une écriture) estobtenu à la fin de l'exécution de l'instruction.

```
var fs = require("fs");

var data = fs.readFileSync("fichier.txt");

console.log("Contenu du fichier fichier.txt : ");

console.log(data);
```

Gestion asynchrone
La gestion asynchrone des fichiers est particulière à Node. Elle permet de ne pas bloquer le programme du serveur lors des opérations de lecture ou d'écriture des fichiers.

```
var fs = require("fs");

fs.readFile("fichier.txt", function(err, data){

console.log("Contenu du fichier fichier.txt : ");

console.log(data);

});

console.log("Suite du programme");
```

> ans le module fs de Node, les méthodes comportant le mot "Sync" dans leur nom sont synchrones, et toutes celles qui ne contiennent pas ce mot sont asynchrones.

9.1/ Ouvrir et fermer un fichier

Méthode	Signification
fs.open(path, flags, callback)	Ouvre le fichier selon le chemin indiqué (path). La fonction de callback est de la forme function (err, fd) dans laquelle fd est le descripteur du fichier ouvert s'il n'y a pas eu d'erreur. Le paramètre flags est une chaîne de caractères indiquant le mode d'ouverture du fichier (lecture, écriture, ajout, etc.). Ces caractères sont décrits à la suite.
fs.openSync(path, flags)	Version synchrone de fs.open(). Le retour de la méthode est le descripteur de fichier obtenu (fd).

Flags utilisés lors de l'ouverture de fichier

Flag	Signification
r	Ouverture du fichier en lecture seule. Le fichier doit exister.
r+	Ouverture du fichier en lecture et écriture. Le fichier doit exister.
w	Ouverture du fichier en écriture seule. Le fichier est créé (s'il n'existe pas) ou écrasé (s'il existe).
w+	Ouverture du fichier en lecture et écriture. Le fichier est créé (s'il n'existe pas) ou écrasé (s'il existe).
a	Ouverture du fichier en mode ajout. Le fichier est créé (s'il n'existe pas) ou modifié (s'il existe).
a+	Ouverture du fichier en lecture et ajout. Le fichier est créé (s'il n'existe pas) ou modifié (s'il existe).

9.2/ Fermer un fichier

Méthode	Signification
fs.close(fd, callback)	Ferme le fichier représenté par son descripteur de fichier fd. La fonction de callback est appelée à l'issue de la fermeture.
fs.closeSync(fd)	Version synchrone de fs.close().

> Un fichier peut être ouvert en utilisant le mode synchrone et fermé dans le mode asynchrone (ou l'inverse).

9.3/ Lire un fichier

Méthode	Signification
fs.readFile(filename, [options], callback)	Lit le fichier indiqué, puis appelle la fonction de callback de la forme function(err, data), dans laquelle le paramètre data représente le contenu du fichier lu. Le paramètre options permet de préciser le mode d'ouverture (flag) du fichier et son encodage (encoding). - L'option flag peut valoir "r", "r+", "w", "w+", "a" ou "a+" (voir tableau 6-2). Par défaut, elle a la valeur "r" (lecture seule). - L'option encoding peut valoir "utf8" ou null. Par défaut, elle vaut null, ce qui retourne le contenu du fichier sous forme de buffer d'octets.
[options])	Version synchrone de fs.readFile(). Le contenu du fichier est retourné par la fonction.

```
var fs = require("fs");

fs.readFile("fichier.txt", { encoding : "utf8" }, function(err, data){

console.log(data);

});
```

9.4/ Écrire dans un fichier

Il existe différentes façons d'écrire dans un fichier :

- soit on écrase son précédent contenu, s'il existe ;
- soit on ajoute en fin de fichier de nouveaux octets.

Méthode	Signification
fs.writeFile(filename, data, callback)	Remplace le contenu du fichier par les octets (Buffer ou String) indiqués dans le paramètre data. Si data représente une chaîne, elle est encodée en UTF-8. La fonction de callback est de la forme function(err) et elle est appelée à la fin de l'écriture dans le fichier. Le fichier est créé s'il n'existe pas, sinon il est écrasé.
fs.writeFileSync(filename, data)	Version synchrone de fs.writeFile().

```
//Écrire dans un fichier, puis le relire

var fs = require("fs");

fs.writeFile("fichier.txt", "Bonjour éric", function(err) {

  fs.readFile("fichier.txt", { flag : "r", encoding : "utf8" }, function(err,data) {

    console.log(data);

  });
});
```

9.5/ Dupliquer un fichier

Méthode	Signification
fs.link(srcpath, destpath, callback)	Copie le fichier de l'emplacement source vers l'emplacement destination. La fonction de callback est de la forme function(err) dans laquelle err vaut null si le fichier a été correctement copié. Le fichier destination ne doit pas déjà exister avant la copie, sinon une erreur se produit.
fs.linkSync(srcpath, destpath)	Version synchrone de fs.link(). Provoque une exception si le fichier ne peut pas être copié.

```
//Dupliquer le fichier fichier.txt en fichier2.txt

var fs = require("fs");

fs.link("fichier.txt", "fichier2.txt", function(err) {

if (err) console.log(err);

else console.log("Fichier copié");

});
```

9.5/ Supprimer un fichier

Méthode	Signification
	back est de la forme function(err) dans laquelle err vaut null si le fichier a été correctement supprimé.
fs.unlinkSync(path)	Version synchrone de fs.unlink(). Provoque une exception si le fichier ne peut pas être supprimé.

```
//Supprimer le fichier fichier.txt

var fs = require("fs");

fs.unlink("fichier.txt", function(err) {

if (err) console.log(err);

else console.log("Fichier supprimé");

});
```

9.6/ Supprimer un fichier

Méthode	Signification
fs.rename(oldpath, newpath, callback)	Renomme le chemin (fichier ou répertoire) oldpath en newpath. La fonction de callback est de la forme function(err) dans laquelle err vaut null si la modification de nom est effectuée. Si le chemin d'origine correspond à un fichier, et si le fichier de destina- tion existe, ce dernier est écrasé. Si le chemin d'origine correspond à un répertoire, et si le répertoire de destination existe, rien n'est modifié.
fs.renameSync(oldpath, newpath)	Version synchrone de fs.rename(). Provoque une exception si la modification de nom ne peut être effectuée.

```
//Renommer le fichier fichier.txt en fichier2.txt

var fs = require("fs");

fs.rename("fichier.txt", "fichier2.txt", function(err) {

console.log("Fichier renommé");

});
```

9.7/ Créer ou supprimer un répertoire

Méthode	Signification
fs.mkdir(path, callback)	Crée le répertoire indiqué. La fonction de callback est de la forme function(err) dans laquelle err vaut null si le répertoire a été correctement créé.
fs.mkdirSync(path)	Version synchrone de fs.mkdir(). Provoque une exception si le répertoire ne peut pas être créé.
fs.rmdir(path, callback)	Supprime le répertoire indiqué. La fonction de callback est de la forme function(err) dans laquelle err vaut null si le répertoire a été correctement supprimé. Attention : le répertoire doit être vide (ni fichier ni répertoire), sinon il ne sera pas supprimé.
fs.rmdirSync(path)	Version synchrone de fs.rmdir(). Provoque une exception si le répertoire ne peut pas être supprimé.

```
//Créer le répertoire newdir dans le répertoire courant

var fs = require("fs");

fs.mkdir("newdir", function(err) {

if (err) console.log(err);

else console.log("Répertoire créé");

});

//Supprimer le répertoire newdir précédemment créé

var fs = require("fs");

fs.rmdir("newdir", function(err) {

if (err) console.log(err);

else console.log("Répertoire supprimé");

});
```

9.8/ Lister tous les fichiers d’un répertoire

> Le répertoire courant est symbolisé par “.”, son parent par “..”.

Méthode	Signification
fs.readdir(path, callback)	Appelle la fonction de callback de la forme function(err, files) dans laquelle files est un tableau des noms de fichiers et de répertoires du chemin indiqué.
fs.readdirSync(path)	Version synchrone de fs.readdir(). Le tableau files est retourné par la méthode.

```
//Afficher la liste de tous les fichiers et répertoires du répertoire courant

var fs = require("fs");

fs.readdir(".", function(err, files) {

console.log(files);

});
```

9.8/ Tester l’existence d’un fichier ou d’un répertoire

> Le répertoire courant est symbolisé par “.”, son parent par “..”.

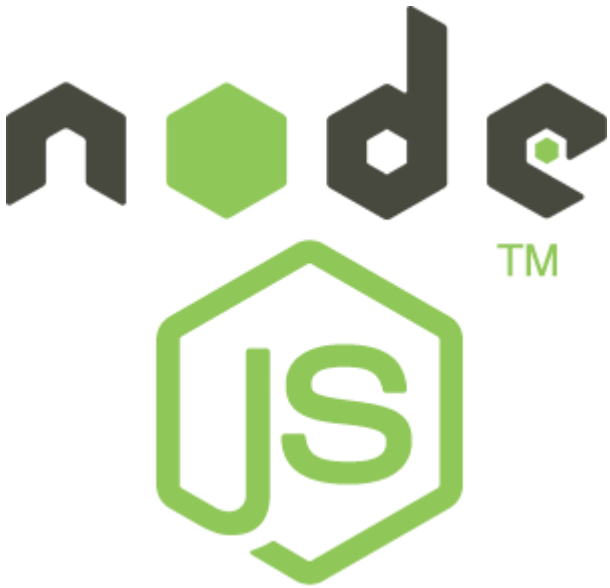
Méthode	Signification
fs.exists(path, callback)	Teste l’existence du fichier ou du répertoire indiqué, et appelle la fonc- tion de callback de la forme function(exists) dans laquelle exists vaut true si le fichier ou répertoire existe, sinon false. Attention : la fonction de callback ne possède pas le paramètre err tra- ditionnel.
fs.existsSync(path)	Version synchrone de fs.exists(). Retourne true si le chemin path existe, false sinon.

```
//Indiquer si un fichier existe

var fs = require("fs");

fs.exists("fichier.txt", function(exists) {

if (exists) console.log("Le fichier fichier.txt existe");

else console.log("Le fichier fichier.txt n’existe pas");

});
```

Ex 13 : Manipulation de fichier

CORRIGÉ

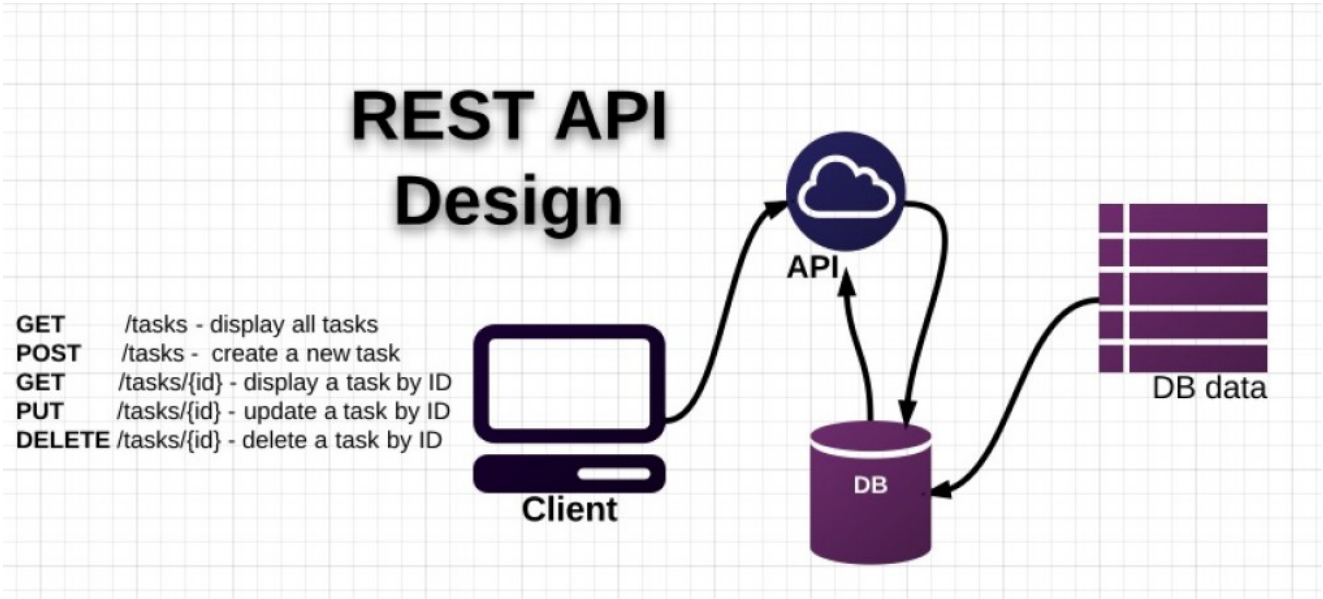
```
En utlisant l'approche Evénementielle,
Créez un module nommé 'database'
sauvegardant la liste dans des fichier json

d'abord de façon synchrone puis asynchrone
```



Gestion des routes

10/ La gestion de routes avec HTTP

10.1/ La gestion de routes avec HTTP



La création d'un serveur HTTP s'effectue au moyen dela méthode **http.createServer()**, définie dans le module http de Node.

Cette méthode retourne un objet server de classe **http.Server**, sur lequel on utilise la méthode **server.listen(port)** permettant de mettre en attente le serveur sur un numéro de port particulier.

Méthode	Signification
http.createServer([requestListener])	Retourne un objet de classe http.Server correspondant au serveur HTTP créé par Node (objet server). La fonction de callback indiquée est optionnelle et permet de traiter l'événement request indiquant qu'un client s'est connecté au serveur.
server.listen(port, [callback])	Met le serveur à l'écoute du port indiqué. La fonction de callback est optionnelle et elle est déclenchée lorsque le serveur est à l'écoute des connexions des clients (la fonction de callback correspond à l'événe- ment listening).

```
//Créer un serveur HTTP affecté au port 3000

var http = require("http");

var server = http.createServer();

server.on("request", function(request, response) {

var html = "";

html += "<html><head><meta charset=utf-8></head>";

html += "<body><p>Bienvenue sur le serveur HTTP de Node !</p></body>";

html += "</html>";

response.end(html);

});

server.listen(3000);
```

10.2/ Gestion des requêtes

> Le traitement de l'événement request, déclenché lorsqu'un utilisateur se connecte au serveur, correspond à une fonction de callback qui prend les deux paramètres.

- Le paramètre **request** est de classe http.IncomingMessage et correspond aux arguments transmis par le client HTTP afin que le serveur puisse traiter sa requête. *C'est également un stream en lecture de classe stream.Readable.*
- Le paramètre **response** est de classe http.ServerResponse et contiendra la réponse retournée au client HTTP. *C'est également un stream en écriture de classe stream.Writable.*

Propriété	Signification
request.url	URL pour laquelle l'événement request est déclenché. Elle correspond à ce qui suit le nom du serveur et le port. Attention : comme l'événement request est appelé également pour le fichier /favicon.ico, il faudra tester si l'URL est différente de ce fichier afin d'effectuer une seule fois le traitement.
request.method	Méthode associée à la requête ("GET", "POST", "PUT", "DELETE"). Si l'URL est introduite directement dans la barre d'adres- ses du navigateur, la requête est "GET").
request.headers	En-têtes (headers) utilisés dans la requête reçue.
request.client	Objet de classe net.Socket correspondant au client TCP qui se connecte au serveur.

10.3/ Gestion des réponses

> La construction de la réponse se fait ici en HTML et elle est transmise au moyen de la méthode **response.end()**

Méthode	Signification
response.write(chunk, [encoding], [callback])	Écrit le buffer d'octets indiqué par chunk sur le stream. Si le buffer est représenté par une chaîne de caractères, le paramètre facultatif encoding permet de préciser l'encodage utilisé ("utf8" par défaut). Le paramètre callback représente une éventuelle fonction de call- back qui sera appelée à la fin de l'exécution de l'instruction.
response.end([chunk], [encoding], [callback])	Ferme le stream. Le buffer chunk, si indiqué, sera écrit sur le stream avant sa fermeture. Les paramètres chunk, encoding et callback sont facultatifs et ont la même signification que dans la méthode write(chunk, encoding, callback). Cette instruction est obligatoire dans le cas d'un serveur HTTP.

> On pourra ainsi utiliser autant d'instructions write() que souhaitées, qui permettront d'envoyer la réponse au navigateur.

10.4/ Fichier statique en réponse

> Cette possibilité permet de modifier le fichier HTML sans avoir à corriger le code du serveur, ce qui permet de produire un affichage différent dans la réponse envoyée au navigateur.

```
//Utilisation de la méthode pipe() pour écrire le flux dans la réponse

var http = require("http");

var util = require("util");

var fs = require("fs");

var server = http.createServer(function(request, response) {

console.log(request.url);

var index = fs.createReadStream("index.html");

index.pipe(response);

});

server.listen(3000);
```

10.5/ Gestion des entêtes

Méthode	Signification
response.writeHead(statusCode, headers)	Indique le code retour de la requête HTTP dans le paramètre statusCode et les en-têtes dans le tableau headers. Le tableau headers est un tableau d'objets dont les clés sont les noms des en-têtes (par exemple, { "Content-Type" : "text/html" }).
response.setHeader(name, value)	Positionne l'en-tête name à la valeur value. Très pratique si l'on veut posi- tionner un en-tête sans l'envoyer immédiatement (à utiliser lorsque la méthode writeHead() provoque une erreur si elle est appelée plusieurs fois).

```
var http = require("http");
var util = require("util");
var fs = require("fs");
var url = require("url");
var server = http.createServer(function(request, response) {

console.log(request.url);

var filename = url.parse(request.url).pathname;

if (filename == "/favicon.ico") return;

if (filename == "/") filename = "/index.html";

filename = "." + filename;

fs.exists(filename, function(exists) {

if (!exists) {

filename = "404.html";

response.writeHead(404, {"Content-Type" : "text/html"} );

}

else response.writeHead(200, {"Content-Type" : "text/html"} );

var file = fs.createReadStream(filename);

file.pipe(response);

});

});

server.listen(3000);
```

10.6/ Événements et méthodes gérés par le serveur HTTP

Événement	Signification
request	Un client HTTP a effectué une requête au serveur. La fonction de callback de traitement de l'événement prend les deux paramètres request et response définis précédemment.
close	Le serveur a effectué server.close() et chacun des clients a également clôturé sa connexion avec lui. Une fois que l'événement close a été émis avec ces deux conditions, le serveur ne peut plus recevoir de nouvelles connexions. Remarquons que tant qu'un client n'a pas fermé sa connexion avec le serveur, d'autres clients peuvent toujours se connecter même si le serveur a effectué server.close().

Méthode	Signification
server.listen(port, [callback])	Met le serveur à l'écoute du port indiqué. La fonction de callback est optionnelle et elle est déclenchée lorsque le serveur est à l'écoute des con- nexions des clients (la fonction de callback correspond à l'événement listening).
server.close ([callback])	Effectue la fermeture du serveur HTTP, qui n'acceptera de nouvelles con- nexions que si toutes les connexions déjà ouvertes seront fermées par les clients HTTP. La fonction de callback, optionnelle, sera appelée lorsque l'événement close sera déclenché (voir tableau 10-6).

10.6/ Créer un client HTTP

> Comme pour les serveurs TCP et UDP, il est possible de créer un client HTTP qui se connectera à un serveur HTTP.

Méthode	Signification
http.request(options, callback)	Effectue une requête HTTP au serveur indiqué dans le paramètre options. Retourne un objet de classe http.ClientRequest, qui est un stream en écriture qui servira à envoyer des éléments au serveur (par exemple, un fichier à transférer sur le serveur). La fonction de callback est appelée lorsque la connexion au serveur est effec- tuée avec succès (sinon l'événement error est déclenché sur l'objet retourné par http.request()). Elle possède un paramètre response qui est un stream en lecture permettant de lire la réponse du serveur dans les événements data du stream.

```
//Utiliser la méthode http.request(options, callback)

var http = require("http");

var options = {
hostname: "nodejs.org",
port: 80,
path: "/",
method: "GET"
};

var request = http.request(options, function(response) {

    console.log("**** statusCode : " + response.statusCode);
    console.log("**** headers: " + JSON.stringify(response.headers));
    response.setEncoding("utf8");
    response.on("data", function (chunk) {
    console.log("**** Données reçues : " + chunk);

  });

});

request.on("error", function(e) {
  console.log("Erreur dans http.request() : " + e.message);
});

request.end();
```

Option	Signification
options.hostname	Nom de domaine (ou adresse IP) du serveur à accéder. Par défaut, "localhost".
options.port	Port à accéder sur le serveur. Par défaut, 80.
options.path	Chemin de l’URL à accéder sur le serveur. Par défaut, "/", soit la racine du site.
options.method	Méthode utilisée pour communiquer avec le serveur ("GET", "POST", "DELETE" ou "PUT"). Par défaut, "GET".

Ex 15 : Gestion des routes

CORRIGÉ

cf. §4p79-83

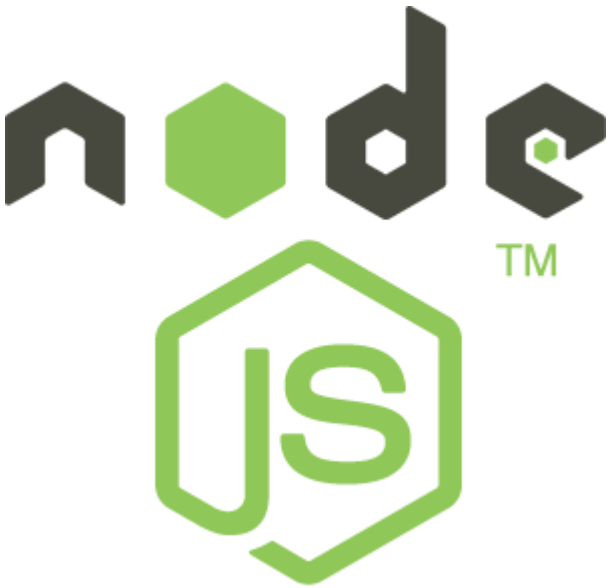
cf. §10

cf. [url (https://fr.wikipedia.org/wiki/Representational_State_Transfer)](https://fr.wikipedia.org/wiki/Representational_State_Transfer)

```
Installer postman pour Chrome Canary
```

```
Definir une route principale pour servir l'IHM de l'application
servant la vue statique fournie.

Definir une API  pour les méthode de  module list:

/list/read/id
/list/create
/list/update/id
/list/delete/id
```

```
Definir une API  REST pour les méthode de  module list:
```



Express.js

11/ Refactoriser avec le framework Express.js

Express est une infrastructure web middleware et de routage, qui a des fonctions propres minimes : une application Express n'est ni plus ni moins qu'une succession d'appels de fonctions middleware.

cf. [Express.js (http://expressjs.com/)]

11.1/ Installation

Créez un répertoire pour héberger votre application et faites-en votre répertoire de travail.

> Utiliser le 'scaffolder express' dans un nouveau dossier de projet

L'outil de générateur d'applications, express, pour créer rapidement un squelette d'application.

```
npm install express-generator -g
```

Affichez les options de commande à l'aide de l'option -h :

```
$ express -h

  Usage: express [options][dir]

  Options:

    -h, --help          output usage information
    -V, --version       output the version number
    -e, --ejs           add ejs engine support (defaults to jade)
        --hbs           add handlebars engine support
    -H, --hogan         add hogan.js engine support
    -c, --css <engine>  add stylesheet <engine> support (less|stylus|compass|sass) (de
        --git           add .gitignore
    -f, --force         force on non-empty directory

$ express myapp
```

Par exemple, ce qui suit crée une application Express nommée myapp dans le répertoire de travail en cours :

> Analyser les fichier générés. Pui installez les dépendances.

```
npm install
```

L’application générée possède la structure de répertoire suivante :

```
.
├── app.js
├── bin
│   └── www
├── package.json
├── public
│   ├── images
│   ├── javascripts
│   └── stylesheets
│       └── style.css
├── routes
│   ├── index.js
│   └── users.js
└── views
    ├── error.jade
    ├── index.jade
    └── layout.jade
```

11.2/ Routage

Le **Routage** fait référence à la détermination de la méthode dont une application répond à une demande client adressée à un noeud final spécifique, c'est-à-dire un URI (ou chemin) et une méthode de demande HTTP (GET, POST, etc.).

Chaque route peut avoir une ou plusieurs fonctions de gestionnaire, qui sont exécutées lorsque la route est mise en correspondance.

La définition de la route utilise la structure suivante :

> app.METHOD(PATH, HANDLER)

- app est une instance d'express.
- METHOD est une méthode de demande HTTP.
- PATH est un chemin sur le serveur.
- HANDLER est la fonction exécutée lorsque la route est mise en correspondance.

```
var express = require('express');
var app = express();

app.get('/', function (req, res) {
  res.send('Hello World!');
});

app.post('/', function (req, res) {
  res.send('Got a POST request');
});

app.put('/user', function (req, res) {
  res.send('Got a PUT request at /user');
});

app.delete('/user', function (req, res) {
  res.send('Got a DELETE request at /user');
});
```

11.2/ Fichiers Statiques

Pour servir des fichiers statiques tels que les images, les fichiers CSS et les fichiers JavaScript, utilisez la fonction de logiciel intermédiaire intégré express.static dans Express.

```
app.use(express.static('public'));
```

11.3/ Utilisation de middleware

Les fonctions de middleware sont des fonctions qui peuvent accéder à l'objet **Request (req),** l'objet **response (res)** et à la fonction middleware suivant dans le cycle demande-réponse de l'application. La fonction middleware suivant est couramment désignée par une variable nommée **next.**

Les fonctions middleware effectuent les tâches suivantes :

- Exécuter tout type de code.
- Apporter des modifications aux objets de demande et de réponse.
- Terminer le cycle de demande-réponse.
- Appeler la fonction middleware suivant dans la pile.

> Si la fonction middleware en cours ne termine pas le cycle de demande-réponse, elle doit appeler la fonction next() pour transmettre le contrôle à la fonction middleware suivant. Sinon, la demande restera bloquée.

Une application Express peut utiliser les types de middleware suivants :

- Middleware niveau application
- Middleware niveau routeur
- Middleware de traitement d'erreurs
- Middleware tiers

11.4/ Middleware niveau application

Liez le middleware niveau application à une instance de l'objet app object en utilisant les fonctions **app.use() et app.METHOD()**, où METHOD est la méthode HTTP de la demande que gère la fonction middleware (par exemple GET, PUT ou POST) en minuscules.

```
var app = express();

app.use(function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});
```

11.5/ Middleware niveau routeur

Le middleware niveau routeur fonctionne de la même manière que le middleware niveau application, à l'exception près qu'il est lié à une instance de express.Router().

```
var app = express();
var router = express.Router();

router.use(function (req, res, next) {
  console.log('Time:', Date.now());
  next();
});

app.use('/', router);
```

11.6/ Middleware de traitement d'erreurs

Définissez les fonctions middleware de traitement d'erreurs de la même façon que d'autres fonctions middleware, à l'exception près qu'il faudra 4 arguments au lieu de 3, et plus particulièrement avec la signature (err, req, res, next)) :

```
app.use(function(err, req, res, next) {
  console.error(err.stack);
  res.status(500).send('Something broke!');
});
```

> Le middleware de traitement d'erreurs comporte toujours quatre arguments. Vous devez fournir quatre arguments pour l'identifier comme une fonction middleware de traitement d'erreurs.

11.7/ Middleware tiers

Utilisez un middleware tiers pour ajouter des fonctionnalités à des applications Express.

Installez le module Node.js pour la fonctionnalité requise, puis chargez-le dans votre application au niveau application ou au niveau router.

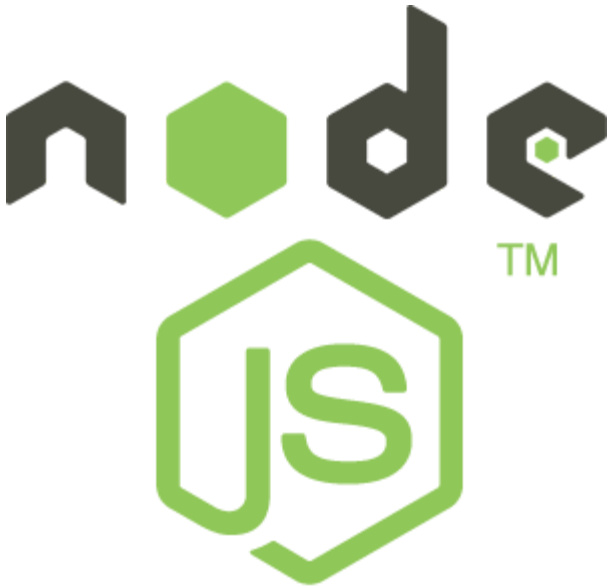
```
var express = require('express');
var app = express();
var cookieParser = require('cookie-parser');

app.use(cookieParser());
```

> Middleware tiers (http://expressjs.com/fr/resources/middleware.html)

Ex 16 : Développer avec Express.js

CORRIGÉ

```
Refactoriser votre application en vous appuyant sur Express.js
```



Persistance des données

12/ Persistance des données

Module clients pour bases de données

> Il existe des modules pour se connecter au SGBD les plus courants.

- node-mysql
- ode-oracledb
- node-mssql
- mongoose
- ioredis
- node_redis
- pg (Postgres)

> Exemple avec SQLite.

```
var sqlite3 = require('sqlite3');

var db = new sqlite3.Database(':memory:');

db.serialize(function() {
    db.run("create table users (login, name)");

    var stmt = db.prepare("insert into users values (?, ?)");
    var users = [ { login: 'pierre',  name: 'Pierre' },
                  { login: 'paul',    name: 'Paul' },
                  { login: 'jacques', name: 'Jacques' } ];

    for (var i in users) {
        stmt.run(users[i].login, users[i].name);
    }

    db.each("select login, name from users", function(err, row) {
        console.log(row.login + ": " + row.name);
    });
});
```

12.1/ MongoDB

MongoDB est une base de données NoSQL parmi les plus populaires du monde (le principal concurrent étant CouchDB).

MongoDB est une base de données orientée « documents », c'est-à-dire une structure de données de n'importe quel type.

12.2/ Installation

MongoDB est accessible sur le site http://www.mongodb.org (http://www.mongodb.org). Téléchargez la version qui
correspond à votre système d'exploitation.

Une fois installé, vous pouvez lancer le shell de MongoDB par la commande mongo. Créons notre première base de données, et appelons-la mabase :

```
> use mabase;
switched to db mabase

> db
mabase
```

12.3/ Insertion

```
> db.utilisateurs.save({ nom: 'Pierre', adresse: { voie: 'Avenue des Rues',
ville: 'Rennes' } });

> db.utilisateurs.save({ nom: 'Jacques', adresse: { voie: 'Rue des Avenues',
ville: 'Paris' } });

> db.utilisateurs.save({ nom: 'Paul' });
```

> Nous venons ici d'insérer trois documents dans une collection.

12.4/ Recherche

> find(critère, projection)

Utiliser **find()** pour retourner toute la liste de la collection utilisateurs.

```
> db.utilisateurs.find();
{ "_id" : ObjectId("5238b1c15fe1afba9cec2027"), "nom" : "Pierre","age" : 18, "adresse"
{ "_id" : ObjectId("5238b1c95fe1afba9cec2028"), "nom" : "Jacques", "adresse" : { "voie
{ "_id" : ObjectId("5238b1d05fe1afba9cec2029"), "nom" : "Paul" }

> db.utilisateurs.find({ nom: 'Pierre' });
{ "_id" : ObjectId("5238b1c15fe1afba9cec2027"), "nom" : "Pierre","age" : 5, "adresse"

> db.utilisateurs.find({ 'adresse.ville': 'Paris' });
{ "_id" : ObjectId("5238b1c95fe1afba9cec2028"), "nom" : "Jacques","age" : 45, "adresse
```

```
db.utilisateurs.find()
```

Pour récupérer les utilisateurs dont l'age = 5

```
db.utilisateurs.find({age:5})
```

On peut aussi utiliser des expressions régulières. Par exemple, tous les prénoms commençant par "j"

```
db.utilisateurs.find({prenom: /^j*/})
```

Pour récupérer les utilisateurs dont l'age est supérieur à 40

```
db.utilisateurs.find({age:{$gt:40}})
```

$gt est un mot clef de mongo qui veut dire greater than (supérieur à). Pour voir la liste complète c'est ici.

Pour récupérer un seul élément (le premier) , utiliser findOne

```
db.utilisateurs.findOne({age:{$gt:40}})
```

Pour récupérer les utilisateurs dont l'âge est 5 ou 10 :

```
db.utilisateurs.find({age:{$in:[5,10]}})
```

Pour récupérer uniquement certaine clé, on utilise l'argument projection de find(). Par exemple, récupérer uniquement les noms des utilisateurs dont l'âge est supérieur à 40

```
db.utilisateurs.find({age:{$gt:40}},{"nom":true})
```

Pour limiter le nombre de résultat à 3 :

```
db.utilisateurs.find().limit(3)
```

Pour ordonner la liste par âge décroissant. -1 pour décroissant et 1 pour croissant.

```
db.utilisateurs.find().sort(age:-1)
```

12.4/ Mise à jour

Pour mettre à jour un document, nous utilisons la méthode update :

update(query, update, options)

```
>db.utilisateurs.update( { nom: 'Paul' }, { nom: 'Paul', nb: 3 } );

>db.utilisateurs.find( { nom: 'Paul' } );
{ "_id" :ObjectId("5238b1d05fe1afba9cec2029"), "nom" : "Paul", "nb" : 3 }
```

Remplacer tous les prénoms Paul par boby

```
db.utilisateurs.update({"prenom":"Paul"},{$set:{"prenom":"boby"}},{multi:true})
```

Ajoute une clé sexe à tous les utilisateurs

```
db.utilisateurs.update({prenom:"boby"}, {$set:{sexe:"male"}}, {multi:true})
```

Ajoute un patient olivier s'il n'existe pas

```
db.utilisateurs.update({prenom:"olivier"}, {$set:{sexe:"male"}}, {upsert:true})
```

save(document, writeConcern)

La différence avec insert est que save, fait un update du document s'il existe déjà.

```
db.patient.save({"prenom":"jean claude", "nom":"Van Damme"})
```

12.4/ Suppression

> **remove(query,justOne)**

Supprimer tous les utilisateurs qui s'appellent olivier

```
db.utilisateurs.remove({prenom:"olivier"})
```

Supprimer la collection

```
db.utilisateurs.drop()
```

Supprimer la base de donnée

```
use mabase
db.runCommand({dropDatabase: 1});
```

12.5/ MongoDB avec Node.js : Mongoose

Il existe plusieurs modules permettant d'accéder à MongoDB en Node.js. Le module de base est mongodb, qui fournit une interface permettant d'effectuer des opérations sur une base de la même manière que nous venons de le faire avec le shell mongo.

> Le module mongoose est un autre module officiel, qui en se basant sur mongodb introduit une couche supplémentaire facilitant la manipulation des données grâce à une couche ODM (object-document mapper)

Pour installer mongoose : `npm install mongoose.`

> API (http://mongoosejs.com/docs/api.html)

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

// 1- Déclaration du modèle
var utilisateurSchema = mongoose.Schema({
    nom: String,
    nb: Number,
    adresse: {
        voie: String,
        ville: String
    }
});
utilisateurSchema.methods.hello = function() {
    console.log("Bonjour, je m'appelle " +this.nom + " !");
};
var Utilisateur = mongoose.model('utilisateurs', utilisateurSchema);

// 2- Opérations sur les données
var db = mongoose.connection;
db.once('open', function() {
    var pierre = new Utilisateur({
        nom: 'Pierre',
        adresse: {
            voie: 'Avenue des Rues',
            ville: 'Rennes'
        }
    });
    pierre.save(function(err, utilisateur) {
        utilisateur.hello();
        mongoose.disconnect();
    });
});

mongoose.connect('mongodb://localhost/mabase2');
```

> Tutorial interactif en ligne (http://mongly.openmymind.net/tutorial/index). Voir également le dossier documentation.

Ex 17 : Introduction à MongoDB

CORRIGÉ

cf. url (https://www.mongodb.org/)

```
Créer une base de données représentant votre logique de fichier
```

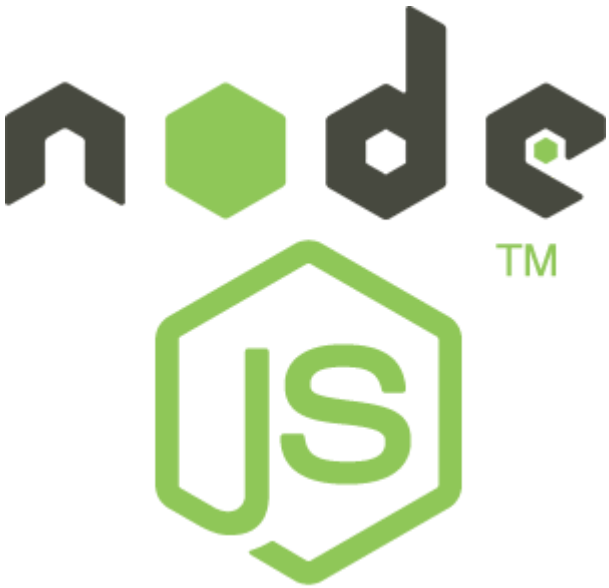
```
Utiliser le client MongoChef pour gérer vos données.
```

```
Utiliser MongoDB Compass pour analyser vos données.
```

Ex 19 : Wrap Up

CORRIGÉ

```
Reliez votre application à sa base de données.
```



Test d’une application Node.js.

13/ Test d'une application Node.js

Avec [Mocha (http://mochajs.org/)](http://mochajs.org/) et [Chai (http://chaijs.com/)](http://chaijs.com/)

```
npm install -g mocha
npm install chai
```

Exemple

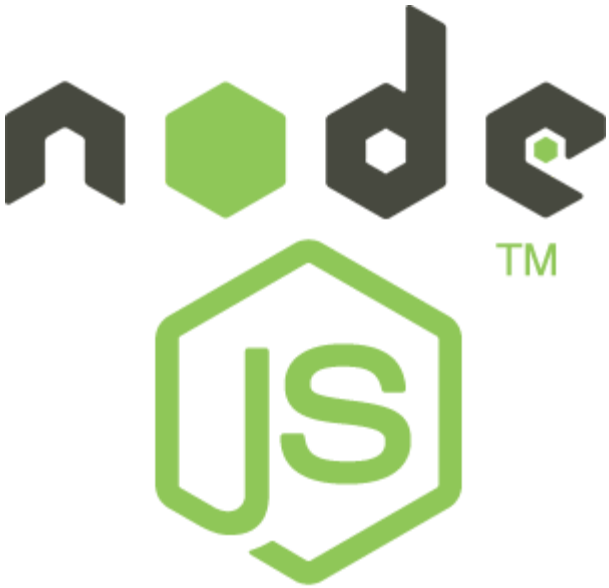
```
//test.js
describe('User', function() {
  beforeEach(function() {
    return db.clear().then(function() {
      return db.save([tobi, loki, jane]);
    });
  });

  describe('#save()', function() {
    it('should save without error', function(done) {
      this.timeout(500);

      var user = new User('Luna');
      user.save(done);
    });

    it('should refuse nameless users');
  });

  describe('#find()', function() {
    it('respond with matching records', function() {
      return db.find({ type: 'User' }).should.eventually.have.length(3);
    });
  });
});
```

```
> mocha test.js
```

Ex 19 : Test

CORRIGÉ

```
Créer un fichier pour tester votre module list
Lancer mocha
```



Annexes.

Annexes.

Installations

Pour les différents systèmes d'exploitation

Certains systèmes d'exploitation fournissent leur propre mouture de Node.
Les procédures les plus courantes sont décrites ci-après.

Une liste complète mais non exhaustive est mise jour sur le wiki du dépôt GitHub de Node à cette adresse : https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager (https://github.com/joyent/node/wiki/Installing-Node.js-via-package-manager).

Si malgré tout votre système n'y était pas listé, le mieux reste encore d'*utiliser un binaire*, de *compiler depuis les sources*.

Linux

Node est disponible dans les dépôts des systèmes suivants :

- *Gentoo* : `emerge nodejs`
- *Ubuntu* >= *12.04*, *Debian* >= *jessie*, *Mint* : `sudo apt-get install nodejs`
- *Fedora* >= *18* : `sudo yum install nodejs npm`
- *Arch Linux* : `pacman -S nodejs`
- *FreeBSD*, *OpenBSD* : `pkg install node`

Si votre système d'exploitation ne dispose pas de paquet pour Node, essayez dans l'ordre :

1. <>;
2. le téléchargement du binaire Node sur son site officiel ;
3. la compilation manuelle de Node.

OS X

OS X ne dispose pas de gestionnaire de paquet par défaut.
Quelques projets populaires permettent toutefois d'y remédier :

- *homebrew* : `brew install node`
- *MacPorts* : `port install nodejs`

Si vous n'utilisez aucun de ces gestionnaires de paquet, vous pouvez essayez :

1. *Node Version Manager* (voir ci-après) ;
2. le téléchargement du binaire Node sur son site officiel ;
3. la compilation manuelle de Node.

Windows

Le moyen le plus simple d'installer Node sous Windows est de *télécharger l'installateur officiel* depuis la page de téléchargements de Node.

Toutefois si vous utilisez déjà un gestionnaire de paquet, voici quelques recommandations :

- *scoop.sh* : `scoop install nodejs`
- *Chocolatey* : `cinst nodejs.install`

Node Version Manager - NVM

Node Version Manager est un logiciel permettant de gérer plusieurs versions de Node en même temps, sur une même machine.
Il est communément abrégé en *nvm*.

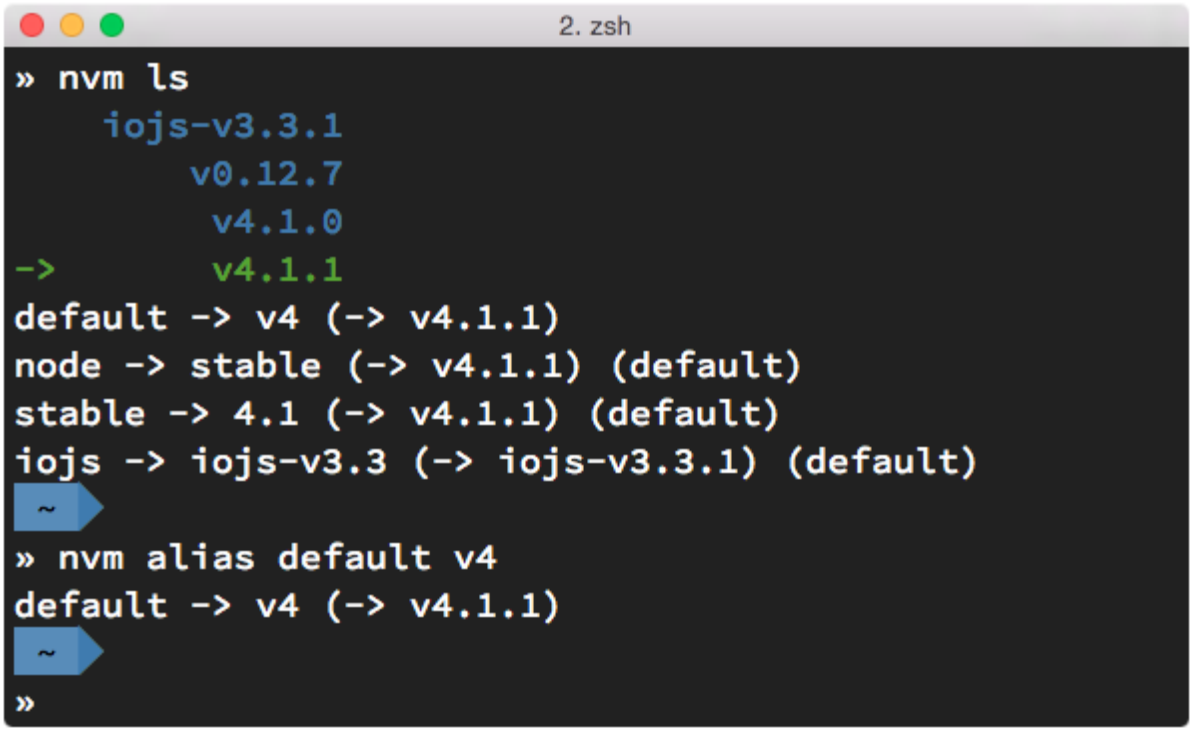
nvm est l'équivalent de *rvm* dans le monde Ruby, de *phpenv* dans le monde PHP ou encore de *virtualenv* pour Python.

Installation de *nvm* et de Node {nodeCurrentVersion} sur un environnement Linux.

```
curl -o- https://raw.githubusercontent.com/creationix/nvm/v0.26.1/install.sh | bash
nvm install v4
nvm alias default v4   <1>
```

<1> La version par défaut est désormais la dernière version stable de Node {nodeCurrentVersion}.

Liste des versions installées de Node.



Les instructions d'installation à jour se trouvent sur https://github.com/creationix/nvm (https://github.com/creationix/nvm).

> *n* est une alternative à *nvm* écrite en… JavaScript.
> Elle a l'avantage d'être compatible avec tous les systèmes d'exploitation compatibles avec le Shell Unix *Bash*. https://www.npmjs.com/n (https://www.npmjs.com/n)

nvm ne fonctionne pas sur les ordinateurs équipés de Windows.
Il existe trois autres alternatives : *nvm-windows*, *nvmw* et *nodist*.

nvmw nécessite d'avoir Git et Python tandis que *nodist* se base uniquement sur Node.
Dans les deux cas, leur installation est très simple.

- https://github.com/coreybutler/nvm-windows (https://github.com/coreybutler/nvm-windows)
- https://github.com/hakobera/nvmw (https://github.com/hakobera/nvmw)
- https://github.com/marcelklehr/nodist (https://github.com/marcelklehr/nodist)

Compiler depuis les sources

Certaines situations exigeront que vous compiliez Node.
Ce sera le cas si vous cherchez à tirer parti au maximum des instructions de votre CPU ou si aucun binaire n'est disponible pour votre plate-forme.

La compilation manuelle requiert la présence de *GCC* 4.2+, de *Python* 2.6+ et de *GNU Make* 3.81+. +
La procédure de compilation ressemble fortement à ceci :

Étapes de compilation de Node

```
curl -sS \
  https://nodejs.org/dist/v4.1.1/node-v4.1.1.tar.gz \
  | tar -zxf -
cd node-v4.1.1
./configure && make && make install
```

Les instructions pouvant varier fortement d'un système d'exploitation à l'autre, consultez les dépendances et instructions complètes à cette adresse
https://github.com/joyent/node/wiki/installation
(https://github.com/joyent/node/wiki/installation).

Outils de développement

Les logiciels présentés dans les pages suivantes couvrent un large spectre de besoins : écriture du code, coloration syntaxique, inspection dynamique, débogage, productivité et intégration à l'écosystème Node.

- Atom : http://atom.io/ (http://atom.io/)
- Brackets : http://brackets.io/ (http://brackets.io/)

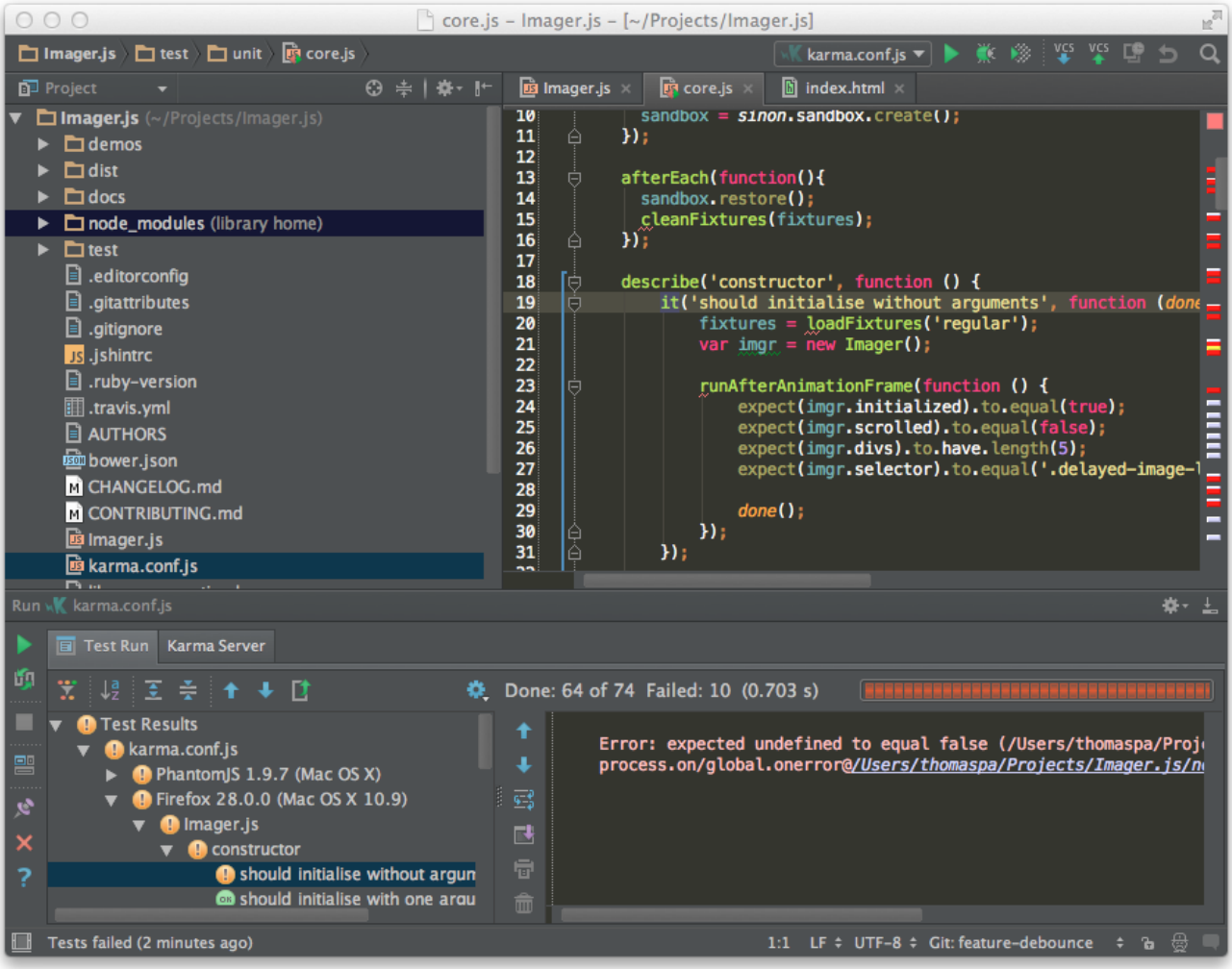
WebStorm

WebStorm est un environnement de développement *(IDE)* commercialisé par la société *JetBrains*.
Il est dédié au développement Web, ce qui inclut JavaScript, Node, mais aussi CSS et HTML.

JetBrains est une entreprise connue dans d'autres écosystèmes pour ses *IDE Pycharm* (pour Python) et *IntelliJ IDEA* (pour Java).

WebStorm est compatible Windows, Linux et OS X.
Ses forces résident dans sa relative légèreté (en comparaison à son concurrent *Eclipse*), une auto-complétion intelligente prenant en compte les modules CommonJS et AMD, une intégration des outils populaires dans l'écosystème Node (npm, JSHint, Mocha, Karma, Grunt, Bower etc.) et un débogage avancé.

> Le site officiel de *WebStorm* met à disposition une documentation lisible et complète ainsi que des vidéos illustrant les fonctionnalités clés du logiciel.
> **Le téléchargement du logiciel inclut une période d'essai de 30 jours.**
> http://www.jetbrains.com/webstorm/ (http://www.jetbrains.com/webstorm/)



L'éditeur de WebStorm, JetBrains, propose une licence gratuite sous réserve d'une contribution active à un projet open source.

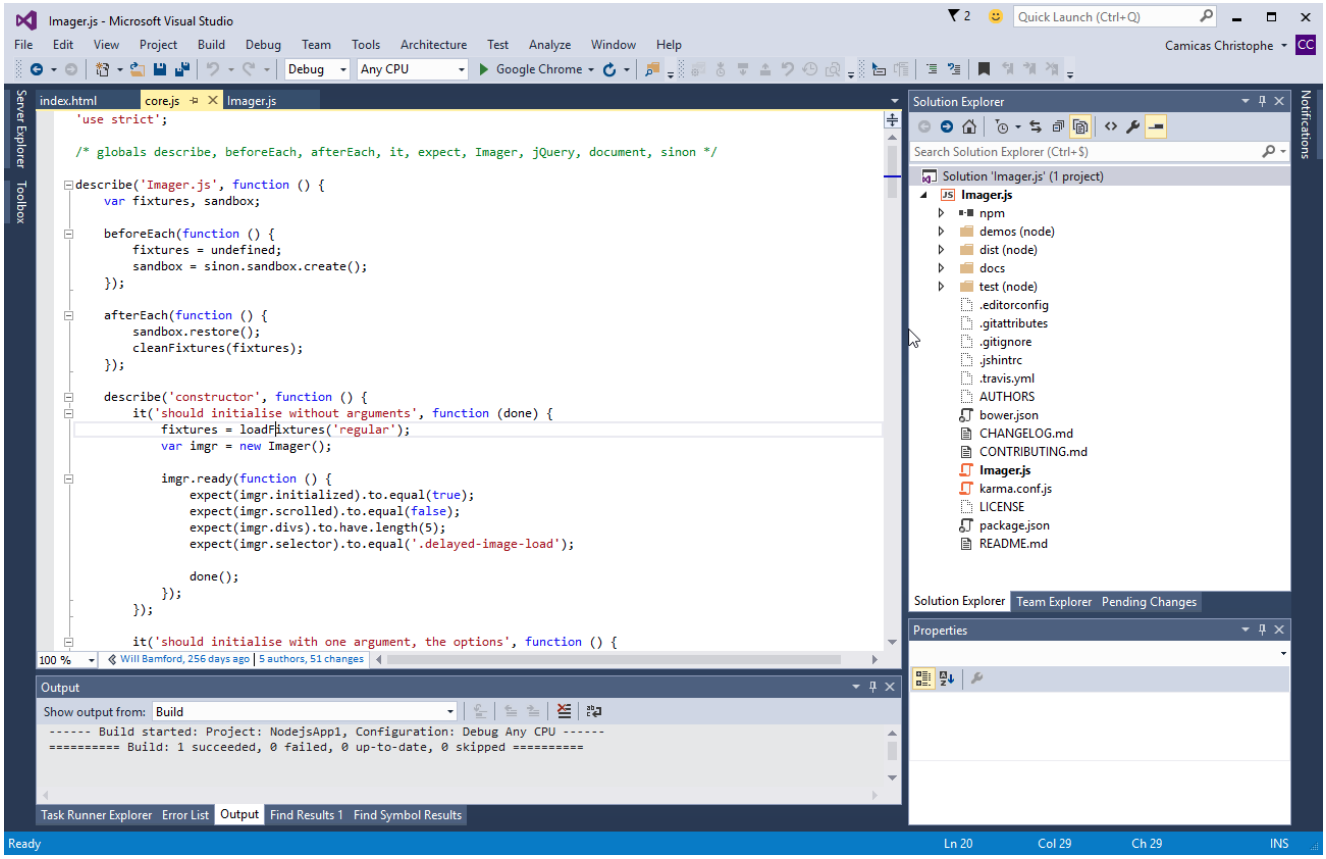
Visual Studio

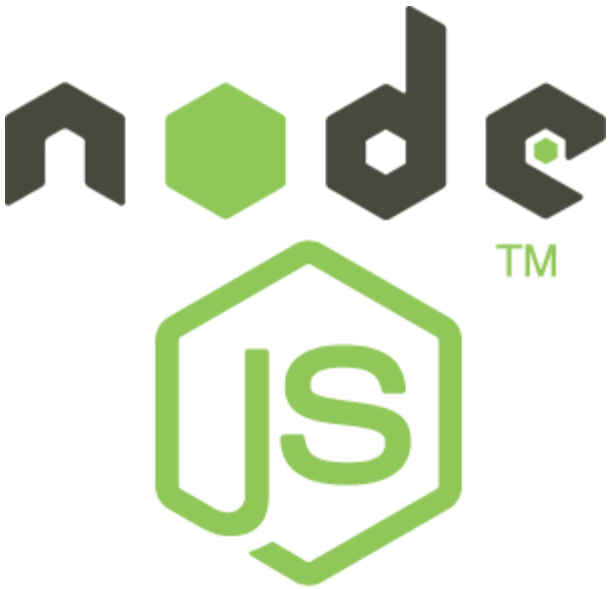
Visual Studio est un environnement de développement *(IDE)* édité par *Microsoft*.
Historiquement dédié au développement sur Windows (Visual Basic, Visual C++), il gère aujourd'hui bien plus de langages (dont C#, HTML, CSS, JavaScript, ASP.Net).
Il est surtout extensible, ce qui permet, entre autre, de lui apporter le support de *Python* ou encore Node.

Visual Studio est uniquement compatible Windows, et existe en plusieurs éditions.
La plus intéressante est la Community Edition.
Elle est gratuite mais permet l'ajout d'extensions.

Afin de bénéficier de l'intégration complète de l'écosystème Node, il est nécessaire d'installer l'extensions *Node.js Tools for Visual Studio* (aussi nommée *NTVS*).
Elle est gratuite, open source et disponible sur GitHub.

- http://www.visualstudio.com/ (http://www.visualstudio.com/)
- https://github.com/Microsoft/nodejstools (https://github.com/Microsoft/nodejstools)


Imager.js - Microsoft Visual Studio
File Edit View Project Build Debug Team Tools Architecture Test Analyze Window Help
Camicas Christophe
Debug
Any CPU
Google Chrome
Quick Launch (Ctrl+Q)
index.html
core.js
Imager.js
'use strict';
/* globals describe, beforeEach, afterEach, it, expect, Imager, jQuery, document, sinon */
describe('Imager.js', function () {
var fixtures, sandbox;
beforeEach(function () {
fixtures = undefined;
sandbox = sinon.sandbox.create();
});
afterEach(function () {
sandbox.restore();
cleanFixtures(fixtures);
});
describe('constructor', function () {
it('should initialise without arguments', function (done) {
fixtures = loadFixtures('regular');
var imgr = new Imager();
imgr.ready(function () {
expect(imgr.initialized).to.equal(true);
expect(imgr.scrolled).to.equal(false);
expect(imgr.divs).to.have.length(5);
expect(imgr.selector).to.equal('.delayed-image-load');
done();
});
});
it('should initialise with one argument, the options', function () {
100 %
Will Bamford, 256 days ago | 5 authors, 51 changes
Solution Explorer
Search Solution Explorer (Ctrl+$)
Solution 'Imager.js' (1 project)
Imager.js
npm
demos (node)
dist (node)
docs
test (node)
.editorconfig
.gitattributes
.gitignore
.jshintrc
.travis.yml
AUTHORS
bower.json
CHANGELOG.md
CONTRIBUTING.md
Imager.js
karma.conf.js
LICENSE
package.json
README.md
Solution Explorer Team Explorer Pending Changes
Properties
Output
Show output from: Build
------ Build started: Project: NodejsApp1, Configuration: Debug Any CPU ------
========== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped ==========
Task Runner Explorer Error List Output Find Results 1 Find Symbol Results
Server Explorer Toolbox
Notifications
Ready
Ln 20 Col 29 Ch 29 INS



A bientôt.