NODE.js

Rev : 03/05/25

NPM

Installation

Installation de l'outil "n" (Install / MAJ de Node)

```
$ curl -o /usr/local/bin/n https://raw.githubusercontent.com/visionmedia/n/master/bin/n
$ chmod +x /usr/local/bin/n
$ n stable
```

Check version :

```
$ node -v
```

Install dernière version :

```
$ n latest
```

Install version précédente :

```
$ n prev
```

Autre manière d'installer sous debian :

```
$ curl -sL https://deb.nodesource.com/setup | bash -
$ sudo apt get install nodejs
```

Sous MidnightBSD :

```
$ cd /usr/mports/www/node && sudo make install clean
```

Gestionnaire de paquet npm

```
$ npm install mon_paquet
```

Certains paquets contiennent des exécutables et sont souvent installés globalement :

```
$ npm install --global mon_paquet
```

check ceux qui peuvent être mis à jour :

```
$ npm outdated --global
```

check un paquet en particulier :

```
$ npm outdated --global paquet_particulier
```

Effectuer une mise à jour :

```
$ npm update --global
$ npm update --global mon_paquet
```

Suppression d'un paquet :

```
$ npm uninstall --global mon_paquet
```

Recherche de paquets sur le répertoire en ligne npmjs.org :

```
$ npm search check password hash
$ npm search graphics
...
```

Numéro de version :

```
<majeur>.<mineur>.<patch>
exemple : 1.2.3

^1.2.3 : toute version sup. ou égale à 1.2.3, mais inf. à 2.0.0
```

Initialisation, gestion des dépendances

```
$ npm init
```

Pour enregistrer des dépendances, il faut convertir un projet en paquet (avec création d'un fichier package.json). C'est ce que fait la commande **npm init**.

Ajouter une dépendance de production :

```
$ npm install --save mon_module
$ npm install --save mon_module@1.2.3
```

Ajouter une dépendance optionnelle :

```
$ npm install --save-optional mon_module_opt
```

Ajouter une dépendance de développement :

```
$ npm install --save-dev mon_module_dev
```

Mise à jour des contraintes de version :

npm update ne fait que récupérer la dernière version d'un paquet qui respecte certaines contraintes.
Pour mettre à jour ces contraintes, on peut utiliser l'outil npm-check-updates :

```
$ npm install --global npm-check-updates
$ npm-check-updates -u
```

Suppressions :

```
$ npm uninstall --save mon_module
$ npm uninstall --save-optional mon_module
$ npm uninstall --save-dev mon_module
```

Listing (liste des paquets installés dans un projet) :

```
$ npm ls
$ npm ls -depth 1
$ npm ls --long
```

NPM LOGIN (après création d'un compte)

```
$ npm login --auth-type legacy
```

(prévoir mot de passe + appli 2FA)

contrôle de l'état de la connexion :

```
$ npm whoami
```

MODULES

2 types de modules coexistent (pour des raisons historiques) avec Node.js :

- Common JS modules
- ES modules(Ecma Script)

les fonctions d'import / export de Common JS ont évoluées avec Ecma Script
La procédure "classique" pour importer un module en JS :

```
<script type="module">
import message from "./message.js";
</script>
```

Dans Node, avec ES modules, chaque fichier est traité comme un module séparé.

exemple (contenu de foo.js) :

```
const circle = require('./circle.js');
console.log(`The area of a circle of radius 4 is ${circle.area(4)}`);
```

Contenu de circle.js :

```
const { PI } = Math;
exports.area = (r) => PI * r ** 2;
exports.circumference = (r) => 2 * PI * r;
```

=> le module circle.js a exporté les fonctions area() et circumference()
Les variables, locales au module (comme ici PI) sont privées.

On peut assigner une nouvelle valeur (comme une variable ou une fonction) à la propriété **modules.export**.

Dans le code suivant, `bar.js` utilise le module "square", qui exporte une classe "Square" :

```
const Square = require('./square.js');
const mySquare = new Square(2);
console.log(`The area of mySquare is ${mySquare.area()}`);
```

Le module "square" est défini dans `square.js`:

```
// Assigning to exports will not modify module, must use module.exports
module.exports = class Square {
  constructor(width) {
    this.width = width;
  }

  area() {
    return this.width ** 2;
  }
};
```

Attention à ne pas oublier le "s" à la fin de "exports" > sinon erreur " x is not a constructor"

Le système de module CommonJS est implémenté dans le module "core module".

Pour CommonJS, l'extension de fichier du module doit être **.js** ou **.cjs**, et s'il y a présence d'un fichier package.json (plus proche parent), il ne doit pas y avoir l'inscription "type" : "module" mais "type" : "commonjs", ou encore, aucun paramètre "type" présent.

L' extension **.mjs** est réservée aux modules **EcmaScript**.

L'extension peut aussi être **.js** si le package.json parent le plus proche contient une ligne avec

"type" : "module" ou s'il ne contient PAS "type" : "commonjs" et qu'il possède la syntaxe propre à ES module.

Exemple de fichiers mjs :

```
// distance.mjs
export function distance(a, b) { return Math.sqrt((b.x - a.x) ** 2 + (b.y -
a.y) ** 2); }
```

```
// point.mjs
export default class Point {
  constructor(x, y) { this.x = x; this.y = y; }
}
```

premier cas : une fonction, deuxième fichier : une classe.

Un module commonJS peut les charger ainsi :

```
const distance = require('./distance.mjs');
console.log(distance);
// [Module: null prototype] {
//   distance: [Function: distance]
// }

const point = require('./point.mjs');
console.log(point);
// [Module: null prototype] {
//   default: [class Point],
//   __esModule: true,
// }
```

Exemple expérimenté en utilisant le module square d'après l'exemple + haut.

```
// fichier bar.js
const Square = require('./square.js'); //module importé : la classe Square.
const readline = require('readline');
const r1 = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});
const laQuestion = new Promise((resolve,reject) => {
  r1.question('côté du carré ? ', cote => {
    console.log(`cote : ${cote}`);
    r1.close();
    if (cote) { resolve(cote) }
    else { reject(new Error("erreur dans la promesse")); }
    });
});

laQuestion.then((cote) => {
    carre = new Square(cote);
    console.log(`Surface du carré : ${carre.area()}`);

});

    if (typeof(cote) != 'undefined') {
    console.log(`Après les fonctions, cote= ${cote}`);
     }
```

>> quand on exécute node bar.js, le programme demande le côté du carré, assigne la valeur saisie à la variable "cote" et finalement affiche la surface du carré calculée.

Attention : la dernière portion du code (if (typeof(cote)...) n'est pas exécutée car la portée de "cote" n'est pas accessible à cette endroit

En modifiant la fin du script comme ceci :

```
laQuestion.then((cote) => {
    carre = new Square(cote);
    console.log(`Surface du carré : ${carre.area()}`);
    return cote;
})
.then ((val) => { console.log(`parseFloat(val) : ${parseFloat(val)}`); if
(parseFloat(val)>10) console.log("cote plus grand que 10") } );
```

on permet, grâce à la valeur de retour (return cote), de transmettre la variable "cote", nommée "val" dans la dernière instruction.

Exemple d'une config pour le développement d'un projet

```
# Vérification de Node.js

node --version

# Si vous voyez une version affichée, Node.js est correctement installé
# Sinon, rendez-vous sur nodejs.org pour l'installer

# Installation globale de TypeScript

npm install -g typescript

# Vérification de l'installation de TypeScript

tsc --version
```

```
# Création et accès au dossier du projet

mkdir mon-projet-typescript

cd mon-projet-typescript

# Initialisation du projet npm

npm init -y

# Installation locale de TypeScript

npm install typescript --save-dev

# Génération du fichier de configuration TypeScript

npx tsc --init
```

Pour un projet TypeScript :

Installation des Dépendances de Base

Installons TypeScript et les outils de développement essentiels :

```
# Dépendances de développement principales

npm install --save-dev typescript           # Le compilateur TypeScript

npm install --save-dev ts-node             # Pour exécuter TypeScript directement

npm install --save-dev nodemon             # Pour le rechargement automatique pendant le développement

npm install --save-dev @types/node         # Types pour Node.js
```

Compilation en Ligne de Commande (tsc)

```
# Installation globale du compilateur
npm install -g typescript

# Compilation d'un fichier
tsc monFichier.ts

# Compilation avec surveillance des changements
tsc monFichier.ts --watch
```

6. Vérification de l'Installation

Pour vérifier que tout fonctionne :

```
# Installation des dépendances
npm install

# Démarrage en mode développement
npm run dev

# Construction du projet
npm run build
```