

# Node JS

PARTIE 03 : GESTIONNAIRE DE PAQUETS NPM

Joachim ZADI  
J4L-TECHNOLOGIES  
septembre 19

# TABLE DES MATIERES

GESTIONNAIRE DE PAQUETS NPM.....	2
Introduction .....	2
1. Qu'est-ce qu'un paquet ?.....	2
2. Exemple d'installation.....	2
3. Registre npmjs.org.....	2
4. Paquet global.....	3
Recherche du bon paquet.....	4
1. Recherche en ligne de commande.....	4
2. Recherche sur npmjs.org.....	4
3. Critères de confiance.....	5
Versionnage .....	6
1. Numéro de version .....	6
2. Contrainte de version.....	6
Gestion des dépendances .....	7
1. Ajout .....	7
2. Mise à jour.....	9
3. Suppression .....	9

# GESTIONNAIRE DE PAQUETS NPM

## INTRODUCTION

L'intérêt de la plate-forme Node est sa grande modularité. Mais justement, comment créer, diffuser, installer et rechercher des modules facilement ? C'est là qu'entre en jeu le gestionnaire de paquets de Node : **npm**. Il est automatiquement installé avec Node et est donc utilisable sans intervention supplémentaire.

Ce chapitre va donc permettre de comprendre d'abord ce qu'est un paquet, pour ensuite être en mesure de travailler avec. L'objectif est double : vous donner tous les outils pour bien utiliser des paquets qui correspondent à vos besoins, mais aussi être en mesure de créer les vôtres ! Vous trouverez de surcroît une explication sur les normes de versionnage pour être en conformité avec les bonnes pratiques.

## 1. QU'EST-CE QU'UN PAQUET ?

Il est important de saisir la différence entre un module et un paquet.

Concrètement, un paquet est un dossier contenant des ressources décrites par un fichier *package.json*. De fait, la plupart des paquets sont des modules, puisque ce sont des bibliothèques qu'il est possible de charger avec la fonction *require()*. Mais ce n'est pas une obligation : certains paquets contiennent seulement des exécutables.

Pour plus d'informations sur les modules, voir le chapitre Concepts.

## 2. EXEMPLE D'INSTALLATION

Illustrons la simplicité d'utilisation de npm pour installer des paquets supplémentaires, avec l'installation du paquet **mon-paquet** qui expose un module Node. L'instruction pour réaliser cela est :

```
npm install mon-paquet
```

Il est maintenant possible de charger le module *mon-paquet* dans un script avec la fonction *require('mon-paquet')*.

À noter que la philosophie de **npm** est d'installer des paquets dans le paquet courant, c'est-à-dire le dossier parent le plus proche contenant un fichier *package.json* (ou le dossier courant si aucun dossier parent ne contient un fichier *package.json*), et pas dans l'intégralité du système. C'est pour cela qu'il n'est pas nécessaire d'avoir les droits d'administration pour installer des paquets supplémentaires.

## 3. REGISTRE NPMJS.ORG

Il existe un catalogue par défaut, ou registre, qui contient tous les modules publics disponibles, développés par la communauté. Il est disponible à l'adresse <http://npmjs.org/>.

Pour illustrer l'importante activité du projet, voici quelques chiffres sur npmjs.org à l'heure où nous écrivons ces lignes : plus de 60 000 paquets disponibles et près de 200 millions de téléchargements le mois dernier.

## 4. PAQUET GLOBAL

Certains paquets contiennent des exécutables et sont généralement installés globalement, ce qui se fait de la façon suivante :

```
npm install --global mon-paquet
```

Pour information, npm lui-même est installé de cette façon.

Plusieurs opérations sont possibles sur les paquets globaux. Pour voir ceux pouvant être mis à jour, il faut exécuter la commande suivante :

```
npm outdated --global
```

Si un paquet est spécifié, alors seul celui-ci est analysé :

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

Pour effectuer la mise à jour :

```
npm update --global
```

À noter que l'exécution de la commande **update** réalise automatiquement **outdated** avant. Sans préciser le nom du paquet, la commande procède à la mise à jour de tous les paquets globaux.

Pour supprimer un paquet (notez la symétrie des commandes d'installation et de suppression) :

```
npm uninstall --global mon-paquet
```

*Si vous êtes sur un système GNU/Linux et que si ne pouvez/voulez pas utiliser le mode administrateur (su ou sudo), vous trouverez des ressources sur la page <https://github.com/sindresorhus/guides/blob/master/npm-global-without-sudo.md>*

# RECHERCHE DU BON PAQUET

Rien que sur le registre standard [npmjs.org](https://npmjs.org), il existe plusieurs dizaines de milliers de paquets. Heureusement, des outils permettent de rechercher ceux qui répondent le mieux à un problème donné et des critères de confiance permettent de départager les différents candidats.

*Certains développeurs tiennent à jour des listes de paquets intéressants. Une des plus connues est <https://github.com/sindresorhus/awesome-nodejs>.*

## 1. RECHERCHE EN LIGNE DE COMMANDE

La première approche, toujours disponible mais aussi la plus rudimentaire, est d'utiliser la commande `npm search <terme>...` En voici un exemple :

npm search check password hash			
NAME	DESCRIPTION	AUTHOR	DATE
bcrypt-password	bcrypt password...	=kemitchell	2019-07-02
nam-oracle	Hash check input...	=kdclaw3	2018-07-30
kpw-chk	Checks passwords...	=zeen3	2018-07-14
ram-cityworks	Hash check input...	=kdclaw3	2016-06-01
havetheybeenpwned	Test if your user's...	=thejameskyle	2019-04-16
joi-drupal-hash	Use Joi to check a...	=wegolookdev...	2016-03-31
node-pbkdf2	Wrapper around...	=louischatriot	2018-03-09
pbkdf2-js	NodeJs...	=alvaropaco	2016-04-13
password-genie	Module designed for...	=munroe7	2016-10-14
bcrypt-nodejs-pass	Simple Bcrypt...	=thenathan30	2017-11-30
mongoose-bcrypt-compare	Utility for...	=joemalski	2015-04-17
hash-password-default	Pretty common code...	=framp	2014-06-06

## 2. RECHERCHE SUR NPMJS.ORG

Le site web du registre propose un système de recherche plus utile que la ligne de commande, notamment par son système de classement.

The screenshot shows the npmjs.org search interface. A search query 'materialize-css' is entered in the search bar. The results page displays the package 'materialize-css' version 1.0.0, which is public and was published a year ago. Key metrics shown include 0 Dependencies, 154 Dependents, and 25 Versions. The package logo is a stylized red 'M'. Below the package details, there is a command to install it via npm: `> npm i materialize-css`. A weekly download chart shows fluctuating activity between 30 and 45 packages per week.

*Il existe des alternatives à npmjs.org, comme <http://nodejsmodules.org> ou encore <http://nodetoolbox.com>. Cependant, celles-ci ne sont pas forcément à jour. Il est préférable de privilégier npmjs.org pour le moment.*

### 3. CRITERES DE CONFIANCE

La popularité d'un paquet sur npmjs.org est un bon critère mais ce n'est pas le seul.

#### A. POPULARITE SUR GITHUB

Tout d'abord, la plupart des paquets étant développés sur GitHub, il est possible de regarder leur popularité sur ce site qui est beaucoup plus utilisé et donc plus représentatif de la communauté des développeurs.

The screenshot shows a GitHub repository page for 'fgnass / materialize'. At the top right, there are metrics: Watch (1), Star (2), Fork (0). A red box highlights the 'Star' and 'Fork' counts. Below, a timeline shows commits from June 2014. A red box highlights the commit 'tgnaas add build status' (2) and its date '5 years ago'.

Les métriques comme le nombre de favoris ou de forks (projets dérivés) (1) sont des excellents indicateurs de popularité.

Il est également important de regarder si une version est toujours en développement, et pour cela rien de tel que de consulter la date du dernier changement (2).

#### B. INTEGRATION CONTINUE

Un autre critère important de confiance dans la robustesse du code du paquet est la validation des tests unitaires (voir le chapitre Tests).

La plupart des modules importants et reconnus dans la communauté utilisent l'intégration continue, c'est-à-dire que chaque nouvelle version est automatiquement testée afin de garantir un haut niveau de qualité.

Un badge indiquant l'état de la dernière validation est souvent présent dans le fichier README du paquet.

The screenshot shows the Bluebird README page. It features a 'Read Me' button, a 'build failing' badge, and a 'then' badge. The page also includes sections for 'Introduction', 'Topics', and a list of features.

Bluebird is a fully featured promise library with focus on innovative features and performance.

**Topics**

- Features
- Quick start
- API Reference and examples
- What are promises and why should I use them?
- Questions and issues
- Error handling

Ici par exemple, il indique qu'il y a eu des erreurs. Cela n'est pas nécessairement un problème majeur et la présence d'un tel badge indique que le ou les auteurs ont à cœur la qualité de leurs paquets.

## VERSIONNAGE

Le versionnage est au cœur du fonctionnement de npm, d'où l'importance de comprendre son principe général. Il est d'ailleurs une bonne pratique dans le monde du développement.

### 1. NUMERO DE VERSION

Un paquet peut exister en plusieurs versions (0.2.2 et 0.2.3 par exemple). Chaque version correspond à l'état du paquet à un instant donné.

Pour assurer un suivi des versions cohérent, npm utilise la convention de versionnage sémantique, dont les détails sont disponibles sur <http://semver.org/>.

Un nom de version se découpe donc ainsi : <majeur>.<mineur>.<patch>.

Tout d'abord, le numéro <patch> est incrémenté quand le paquet reçoit des correctifs, c'est-à-dire des changements mineurs qui n'impactent pas les utilisateurs du paquet en dehors de la correction d'un problème.

Puis, le numéro <mineur> est incrémenté quand le paquet reçoit une nouvelle fonctionnalité mais sans impact sur les fonctionnalités précédentes (elles n'ont donc pas été modifiées).

Enfin, le numéro <majeur> est à son tour incrémenté quand le paquet est modifié en cassant la compatibilité. Cette cassure s'identifie dans l'utilisation du paquet, qui diffère comparativement à la version précédente.

### 2. CONTRAINTE DE VERSION

Une mauvaise gestion des dépendances peut avoir un impact négatif sur un projet (incompatibilité entre deux versions d'une dépendance par exemple). Pour éviter cela, il est important de contrôler la version des dépendances utilisées. Heureusement, npm est doté d'une fonctionnalité permettant d'utiliser la plus récente d'un paquet tout en assurant la compatibilité.

La syntaxe des dépendances utilisée dans le fichier package.json (voir la section Gestion des dépendances pour en savoir plus sur ce fichier) est la suivante :

- 1.2.3 : indique d'utiliser exactement cette version.
- ^1.2.3 : indique d'utiliser une version qui ne casse pas la compatibilité. Dans cet exemple, npm utilisera toute version supérieure ou égale à 1.2.3, mais inférieure à 2.0.0.

Cette dernière syntaxe, avec le caractère ^, fonctionne de la même façon pour tout premier chiffre différent de 0. Ainsi, pour une version en 0.1.2, npm utilisera toute version supérieure ou égale à 0.1.2 mais inférieure à 0.2.0.

*Pour les conditions avancées de gestion des dépendances, voir <http://npmjs.org/doc/misc/semver>.*

# GESTION DES DEPENDANCES

Dès qu'un paquet est utilisé dans un projet, il devient une dépendance. Il est alors nécessaire de la déclarer afin de permettre au programme d'être installé correctement ailleurs que sur le poste courant.

Pour enregistrer les dépendances, il est nécessaire de convertir le projet en paquet (qu'il n'est bien entendu pas obligatoire de publier), c'est-à-dire de créer un fichier package.json.

Cette étape, qui pourrait être fastidieuse, est avantageusement automatisée par la commande **npm init** qui demande à l'utilisateur de renseigner un certain nombre d'informations :

```
λ npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg>` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
package name: (projets) mon-projet
version: (1.0.0)
description:
entry point: (index.js)
test command:
git repository: cessaire de la
keywords: le poste courant
author: Joachim Zadi
license: (ISC)
About to write to C:\Users\Joachim\Desktop\COURS-NODEJS\COURS\projets\package.json:

{
  "name": "mon-projet",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Joachim Zadi",
  "license": "ISC"
}

Is this OK? (yes)
```

Comme on peut le constater, les paquets trouvés dans le répertoire du projet sont détectés et ajoutés automatiquement.

*Cette initialisation est capitale dès le départ de votre projet. En effet elle permet aux dépendances d'être correctement suivies et enregistrées.*

## 1. AJOUT

Il existe différents types de dépendances pour correspondre à chaque besoin.

## A. DEPENDANCE DE PRODUCTION

Le premier type est celui dit de production, nécessaire au fonctionnement du paquet. Pour installer une dépendance de production, il faut utiliser la commande suivante :

```
$ npm install --save mon-module
```

Le fichier package.json contient maintenant la nouvelle dépendance avec sa version actuelle :

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

Pour installer une version particulière d'un paquet, il suffit de préciser la contrainte de version à la fin de son nom après le symbole @ :

```
$ npm install --save mon-module@0.9.5
```

## B. DEPENDANCE OPTIONNELLE

Le deuxième type correspond aux dépendances dites optionnelles. Ce sont des dépendances (assez rarement utilisées) qui peuvent étendre ou améliorer le fonctionnement du paquet mais qui ne sont pas nécessaires.

Par exemple, un paquet contenant une bibliothèque de décodage d'images peut avoir des dépendances optionnelles pour les différents formats supportés.

L'instruction suivante permet d'installer une dépendance de production :

```
$ npm install --save-optional mon-module
```

Le fichier package.json contient alors :

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

En cas d'erreur de récupération du paquet `mon-module`, npm n'arrête pas le processus d'installation, puisque celui-ci est déclaré comme non essentiel au fonctionnement du projet.

## C. DEVELOPPEMENT

Enfin, le dernier type concerne les dépendances de développement, utilisées pour la phase de développement du projet :

```
$ npm install --save-dev mon-module  
"devDependencies": {  
    "mon-module": "^1.2.3"  
}
```

Ce type de paquet n'est pas récupéré par npm lors d'une installation en production. L'intérêt des dépendances de développement est d'inclure des paquets servant à faire du test (unitaire, fonctionnel), du débogage, de la génération de code... et n'ayant pas d'utilité dans un environnement de production.

## 2. MISE A JOUR

Pour mettre à jour les dépendances tout en respectant les contraintes de versions présentes dans le fichier package.json, c'est la commande suivante qui est utilisée :

```
$ npm update
```

Il est possible de limiter la requête au paquet spécifié :

```
$ npm update mon-paquet
```

À noter que la commande *npm update* ne met pas à jour les contraintes de versions, elle ne fait que récupérer la dernière version du paquet les respectant.

Pour mettre à jour les contraintes, il est possible d'utiliser un outil comme **npm-check-updates**.

Voici les commandes pour l'installer et l'utiliser :

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

## 3. SUPPRESSION

Il est utile de pouvoir supprimer facilement un paquet. Le gestionnaire npm dispose évidemment de la commande ad hoc, symétrique à *npm install*.

Pour une dépendance de production :

```
$ npm uninstall --save mon-module
```

Pour une dépendance optionnelle :

```
$ npm uninstall --save-optional mon-module
```

Pour une dépendance de développement :

```
$ npm uninstall --save-dev mon-module
```