Npm

Npm - Généralités

Npm est un **gestionnaire de dépendances.**

L'acronyme signifie "Node Package Manager".

Historiquement utilisé sur des projets Node (ie. JS hors Web), il l'est désormais également sur presque tous les projets de développement Web, y compris Front.

Tous les grands frameworks Front utilisent Node & Npm, car leur outillage est écrit en JS et que l'on utilise Npm pour gérer les dépendances du projet.

Lorsque l'on parle de NPM, on peut évoquer :

- Son site web
- Son registry
- Sa CLI (command line interface)

Npm - Généralités

Le principe général de NPM est de mettre à disposition une gigantesque base de données (le **registry** npm) contenant des librairies JS publiques ou privées.

Chaque librairie doit suivre le versionning **semver**, dont le principe général est le suivant :

- Un numéro de version est sous la forme MAJOR.MINOR.PATCH
- On incrémente le numéro :
  - Majeur lorsque l'on a un changement d'API qui introduit des incompatibilités avec le code utilisant la librairie dans sa version précédente ;
  - Mineur lorsque l'on ajoute de nouvelles fonctionnalités rétro-compatibles avec la version précédente ;
  - Patch lorsque l'on corrige des bugs qui n'introduisent pas d'incompatibilités.

Npm - Généralités

Les intérêts de NPM sont :

- De ne plus avoir besoin d'aller télécharger soi même ses dépendances
- De bien isoler ses sources de ses dépendances (on ne versionne **jamais** une librairie)
- De gérer les dépendances et les incohérences de versions
- D'être certain de récupérer les librairies dans les bonnes versions
- D'être certain d'avoir téléchargé correctement nos librairies (pas de fichiers corrompus)

En bref… l'époque du téléchargement manuel de nos librairies est révolu :)

Npm - En pratique

Dans un projet utilisant NPM, on retrouve deux fichiers principaux :

- package.json
- package-lock.json

Le fichier package.json contient :

- Des informations générales sur notre projet
- Des scripts (alias de commandes)
- La liste des dépendances de notre projet (on peut indiquer des dépendances de production, de développement, ou des dépendances transitives)

Le fichier package-lock.json contient des métadonnées issues de l'installation des packages.

Il existe également un répertoire **node_modules** qui contiendra l'ensemble des librairies téléchargées via npm sur un projet. **Ce dossier doit être ignoré par votre gestionnaire de sources.**

Npm - En pratique

Les commandes principales de Npm sont :

- npm init
- npm install
- npm remove
- npm pack / npm publish
- npx <nom d'un package npm>

Exemples d'installation de dépendances :

- npm install [--production] => installe toutes les dépendances d'un projet
- npm install @angular/core
- npm install @angular/core@latest => équivalent à la commande précédente
- npm install @angular/core@9.1.0 => permet d'obtenir une version spécifique
- npm install -g @angular/cli => Installation d'un package au niveau OS
- npx create-react-app helloworld => Exécution d'un package (ici init. d'une app react) sans l'installer
- npm install typescript --save-dev => Installation d'un package qui servira uniquement côté développement (ne sera pas inclus lors d'un packaging de production)

Npm - En pratique

Lorsque l'on ajoute une dépendance à notre projet, on peut contrôler la plage de versions autorisées lors de l'installation des dépendances sur un autre poste / répertoire (indispensable lorsque l'on travaille en équipe).

Voici les différentes options :

- Fixer une version :

```
"@angular.core": "9.1.0"
```

- Fixer le numéro de version majeur :

```
"@angular.core": "^9.1.0"
```

Autorise les versions 9.1.0 à 9.X.X

- Fixer les numéros de version majeur & mineur :

```
"@angular.core": "~9.1.0"
```

Autorise les versions 9.1.0 à 9.1.X

Il existe des variantes ("<x.x.x", "x.x.x - x.x.x").

Cela fonctionne dans le meilleur des mondes si les développeurs des packages que l'on utilise respectent bien la norme **semver**.

Npm - En pratique

Lorsque l'on installe un package, npm récupère la dernière version autorisée de celui-ci en fonction des règles établies dans le fichier package.json.

Problème : deux développeurs peuvent se retrouver avec des versions différentes des librairies sur un même projet.

Cas simple : une dépendance dont vous avez spécifié une latitude sur le numéro de patch ou mineur.

Plus vicieux : vous avez spécifié des numéros de version précis, mais une dépendance de vos dépendances a été mise à jour sans que vous ne le sachiez.

Encore plus vicieux : le meilleur des mondes n'existe pas et le développeur d'un package se contrefiche de la norme semver (c'est vilain).

Npm - En pratique

As an example, consider package A:

```
{
  "name": "A",
  "version": "0.1.0",
  "dependencies": {
    "B": "<0.1.0"
  }
}
```

package B:

```
{
  "name": "B",
  "version": "0.0.1",
  "dependencies": {
    "C": "<0.1.0"
  }
}
```

and package C:

```
{
  "name": "C",
  "version": "0.0.1"
}
```

Exemple extrait de la documentation NPM

Après un npm i :

```
A@0.1.0
`-- B@0.0.1
    `-- C@0.0.1
```

Si B est mis à jour en 0.0.2, une nouvelle installation donnera l'arbre de dépendances suivant :

```
A@0.1.0
`-- B@0.0.2
    `-- C@0.0.1
```

Npm - En pratique

Pour répondre à ces problématiques, npm génère un fichier package-lock.json

Ce fichier a pour but de figer (d'où "lock") l'arbre de dépendances téléchargées lors d'un npm install.

De cette manière lorsqu'un développeur clone un repository contenant ce fichier package-lock on est certains de télécharger exactement le même arbre de dépendances.

Ce fichier doit être versionné et mis à jour régulièrement (à voir en fonction de votre politique projet).

Yarn

Il existe un projet : YARN.

Ce projet vise à améliorer NPM principalement en terme de rapidité de téléchargement des dépendances puisqu'il s'effectue en parallèle là où npm fonctionne de manière séquentielle.

Pour l'utiliser, il faut l'installer globalement : `npm install -g yarn`

Puis pour installer les dépendances d'un projet : `yarn install`

La différence de perf n'est pas toujours flagrante.