Des packages avec NPM

Les bases — Épisode 4

Un paquet de paquets

Récupérer des données en ligne, identifier l'utilisateur, afficher un menu sympathique, générer un formulaire… Si l'on prend deux applications web, il y a de grandes chances qu'elles aient beaucoup de fonctionnalités en commun. Ce qui fait la différence, c'est leur capacité à répondre aux besoins de l'utilisateur, en combinant ces petites fonctionnalités basiques de façon intelligente.

La philosophie de Node est orientée vers l'écriture de modules simples, réutilisables et ne servant qu'un seul objectif. Par chance, plus le module est simple et utile, et plus il y a de chances qu'une entreprise ou des développeurs sérieux l'ait mis à disposition de la communauté open source.

Ainsi, en tant que développeur JavaScript, vous pouvez vous concentrer sur les éléments qui sont vraiment importants pour l'utilisateur final.

Mais tous ces petits modules, il va falloir les organiser proprement !

Node Package Manager

Comme son nom le laisse entendre, **npm** est un gestionnaire de paquet pour Node. Cet outil est tellement important qu'il est fourni par défaut avec toute installation de Node.

Il va nous permettre de gérer les dépendances de notre application à d'autres paquets, mais nous fournit aussi des fonctionnalités très utiles que nous allons détailler.

Mais moi je suis développeur frontend, pourquoi Node ?

L'un des avantages de Node prônés par ses défenseurs est que cette technologie permet d'utiliser le même langage pour le serveur et pour l'interface. Alors autant utiliser les mêmes outils dès qu'on le peut !

Note : auparavant B**ower** était très utilisé pour le frontend. Cependant, les librairies récentes proposent généralement une installation via npm.

Mon application, un paquet ?

Le mot “paquet” est relativement général. Un paquet (ou librairie, module, ***package***) est un ensemble de code cohérent permettant de répondre à un besoin précis.

Cela englobe des choses très simples/bas niveau et très réutilisables (envoyer une requête HTTP, gérer les dates), comme des applications de grande taille répondant à un besoin spécifique. On considère donc souvent une application web comme un gros paquet, construits à l’aide de paquets plus petits.

Le fichier package.json

Le fichier `package.json` est systématiquement situé à la racine d'un paquet. Il s'agit d'un point clé de notre application, car c'est dans ce fichier que npm va stocker toutes ses dépendances, méta-données, scripts, etc.

node_modules

Enfin, le dossier `node_modules` contient le code des dépendances de notre application, téléchargé automatiquement par npm en fonction du contenu du fichier `package.json`.

Allez, vous êtes prêts pour la pratique, sortez votre terminal !

🔨 Votre propre package.json

Initialiser le package.json

Rien de plus simple pour initialiser votre fichier `package.json`, déplacer vous dans le dossier qui contiendra votre application, et lancer un `npm init`.

```
mkdir mon-super-projet
cd mon-super-projet/
npm init
```

Le script va vous poser quelques petites questions, que vous pouvez ignorer à ce stade. Voilà, vous avez créé votre premier fichier `package.json`, et par la même occasion transformé votre projet en ***package***.

Ajouter un package

Packages officiels npmjs

`npmjs.com` est la plateforme officielle de publication de npm, gérée par la société Npm, inc. Lorsque l'on demande à npm d'installer un package externe, il va dans un premier temps le chercher parmi les packages publiés sur `npmjs.com`.

La plupart d'entre eux sont sûrs, mais attention, n'importe qui peut publier sur npmjs, rien ne dit que la librairie va fonctionner comme vous le souhaitez ! Dans l'idéal, il faut utiliser des librairies répandues ou fournies par un tiers de confiance (entreprise connue, etc.)

Ajouter un package officiel avec une connexion Internet

Installez par exemple **Babel**, un package très utile qui permet de transformer du JavaScript dernière génération en JavaScript lisible par n'importe quel navigateur.

```
npm install @babel/core @babel/cli
ls
```

Voilà, **npm** vient de vous créer un dossier `node_modules/` (si celui-ci n'existait pas déjà), qui contient un sous-dossier `@babel`.

Ajouter un package officiel sans une connexion Internet

Sans connexion Internet, nous allons utiliser un outil appelé `npmbox`. Il s'agit déjà en soit d'un package NPM.

Il vous faut pour cela disposer d'un fichier de type `nom_du_packet.npmbox` puis exécuter cette commande dans le terminal

```
npmunbox babel.npmbox
```

De la même façon, **npm** aura créé un dossier `node_modules` contenant le sous-dossier `@babel` ainsi que toutes les autres dépendances nécessaires au bon fonctionnement de ce packet.

Les options `--save` et `--save-dev`

Et si je supprime ce dossier `node_modules` ? Aïe, vous êtes mal, car en l'état, aucun moyen de savoir quel packages vous avez installé. Quand il y en a plusieurs dizaines, cela devient rapidement gênant. Il faut donc stocker cette liste dans notre fameux `package.json`, avec l'option `**--save**`.

```
npm install @babel-core @babel/cli --save-dev
npm install lodash --save
```

De la même façon, avec `npmbox`

```
npmunbox babel.npmbox --save-dev
npmunbox lodash.npmbox --save
```

Voilà, cette fois Babel et **Lodash** (une librairie avec des fonctions de base très pratiques) sont installées ET listées dans votre package.json

Si vous supprimez le dossier `node_modules`, il suffit de taper `npm install` pour que `npm` récupère tous les packages listés dans le `package.json`.

Malheureusement, cela ne fonctionnera pas **hors-ligne**, il faudra réinstaller à la main les différents packages.

Des `devDependencices` et des `dependencies`

Les `devDependencies` sont créées avec l'option `--save-dev`. Comme leur nom l'indique, elles ne servent qu'au développeur. Il s'agit de toutes les librairies qui vous permettent de mettre en place un environnement de développement efficace, mais qui n'ont aucun intérêt une fois votre code en production.

Par exemple avec Babel, on compile le code une bonne fois pour toute avant de le mettre en ligne. Les librairies de test sont aussi généralement des `devDependencies`.

Les autres dépendances peuvent aussi être utilisées en production. Par exemple, Lodash pourra effectivement être utile pour faire fonctionner notre interface, ce n'est donc pas une dépendance de développement : on utilisera l'option `--save`.

🐙 Modules hébergés sur une autre plateforme

Tous les packages ne sont pas sur npmjs. Parfois, vous pouvez aussi avoir besoin de modifier un package open source, en créant votre propre version (`fork`), sans pour autant vouloir la publier, ou alors trouver une librairie qui vous plaît mais n'est pas sur npmjs.

Dans ce cas, vous pouvez choisir d'installer votre package directement à partir d'un dépôt distant, par exemple GitHub.

```
npm install git+https://github.com/babel/babel.git
```

L'adresse correspond à `git+ADRESSE_DU_PACKAGE.git` (vous expliquez à npm qu'il s'agit d'un dépôt géré par le gestionnaire de version Git).

Cela vous permet aussi de gérer des packages privés, qui ne sont accessibles qu'au sein de votre entreprise.

Gérer les versions

Les semvers

Lors de l'installation, npm cherche automatiquement la dernière version du package qui vous intéresse.

"devDependencies": {
"babel-cli": "^6.26.0"
}

Les packages doivent *en théorie* respecter la norme du **Semantic Versioning**. De façon simplifiée:

- si vous passez de la version `6.26.0` à la version `6.26.1`, il ne va rien se passer : il s'agit d'une correction de bug ou d'une amélioration très anecdotique. Il s'agit d'un **patch**.
- si vous passez de la version `6.26.0` à la `6.27.0`, il y aura certainement de nouvelles fonctionnalités, il s'agit d'une **version mineure**.
- si vous passez de la version `6.26.0` à la version `7.0.0`, attention à la casse ! Il s'agit d'une **version majeure**, et les développeurs ne vous garantissent plus la retro-compatibilité.

Verrouiller les versions

npm respecte la norme du Semantic Versioning, vous ne devriez pas avoir trop de problèmes avec des packages qui cassent lors des mises à jour.

Mais les développeurs ne sont pas parfaits... Le `^` avant le numéro de version signifie que npm doit chercher une version au moins supérieure à la version `6.26.0`, mais non cassante. Par exemple la version `6.28.3` (mais pas la version `7.1.1` , qui est une nouvelle version majeure).

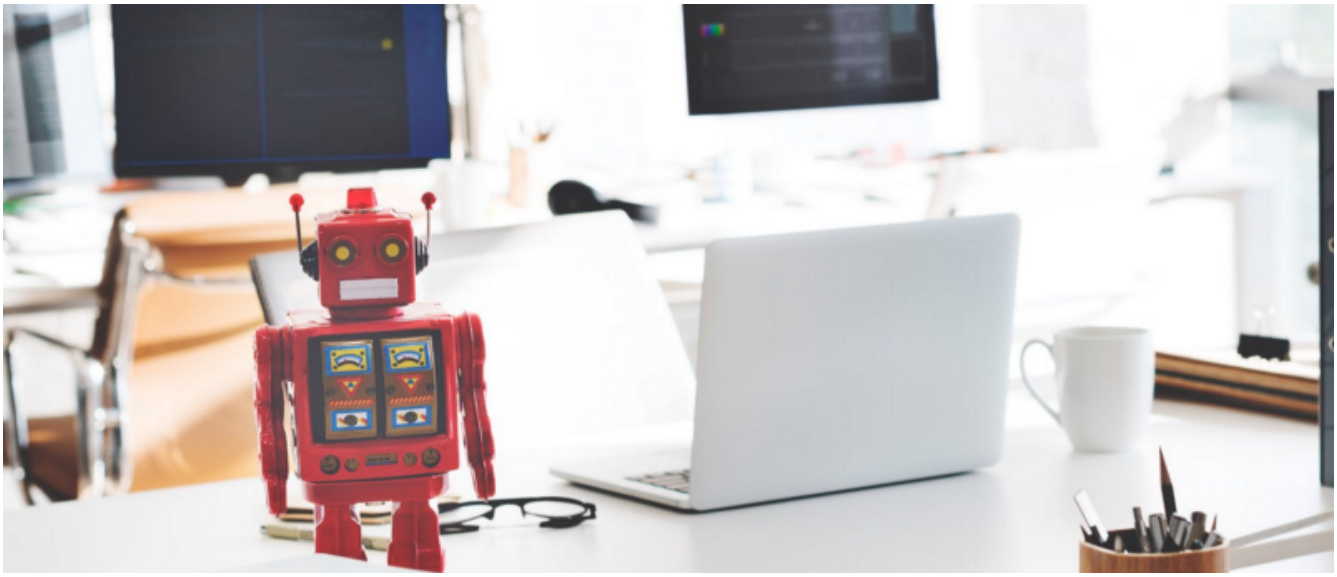
Il s'agit malheureusement d'une mauvaise pratique, il faut idéalement verrouiller les versions, et ne faire les mises à jour que quand vous le souhaitez. Vous pouvez supprimer manuellement le `^` dans le `package.json` :

"devDependencies": {
"babel-cli": "6.26.0"
}

Installer une version spécifique

Pour installer une version spécifique d'un package, il suffit de la préciser lors de l'installation en ajoutant un `@` suivi du numéro de version après le nom du package.

npm show babel-cli versions # pour voir toutes les versions
npm install babel-cli@6.24.1 --save-dev



Des scripts

Créer vos scripts

Un package, ce n'est pas seulement du code, c'est aussi un ensemble de process : lancer l'application localement, l'installer, lancer les tests, le publier, etc.

Votre `package.json` contient ainsi un champ `scripts`, dans lequel vous allez pouvoir... ajouter des scripts.

scripts:{
sayHello: "echo hello"
}

npm run sayHello

Des scripts cross-platform ?

Un problème évident pour les scripts complexes est l'utilisation cross-platform. `**cross-env**` vous permet par exemple de gérer les variables d'environnement (les syntaxes Windows et Linux sont différents sur ce point).

Pour aller plus loin, `**shelljs**` vous permet d'écrire le script en JavaScript, et vous pourrez ensuite l'exécuter avec Node avec un fonctionnement identique sur tous les OS.

Les scripts spéciaux

Hooks

Les `hooks` sont exécutés par npm à des moments précis du cycle de vie de l'installation.

Par exemple, un script nommé `postinstall` sera systématiquement exécuté après un `npm install`. Il peut par exemple permettre d'installer une librairie tierces non JavaScript, ou de télécharger automatiquement la documentation de l'application sur un serveur distant.

Scripts spéciaux et conventions

On retrouve dans tout bon package.json les scripts suivants :

- `start` : démarre l'application

- `build` : compile l'application
- `dev` : démarre le mode développement
- `test` : lance les tests

`start` et `test` sont des scripts spéciaux, `build` et `dev` sont des conventions extrêmement classiques.

Alternatives et outils supplémentaires

Maintenant que vous êtes passé maître dans l'utilisation de `npm`, vous pouvez vous intéresser aux outils complémentaires suivants :

- `yarn`, une alternative à `npm` plus rapide qui connaît un franc succès

Autre avantage méconnu de `_yarn_` : les illustrations de la page d'accueil sont particulièrement géniales

- `npx` permet de lancer localement des packages Node. C'est un peu original, mais pratique par exemple pour créer une appli React avec la dernière version de `create-react-app` sans avoir à installer le package entier.
- `lerna` permet de gérer aisément des applications complexes construites comme des ensembles de packages

Partager son code c'est bien, mais encore faut-il pouvoir l'écrire ! Dans nos prochains articles, nous reviendrons sur les points clés de la syntaxe ES6, qui fait tout le charme du JavaScript moderne.