

NPM

NPM se puede considerar como las siglas de **Node Package Manager**, es decir, gestor de paquetes de **NodeJS**, un entorno de ejecución multiplataforma para ejecutar Javascript no sólo en un navegador web (como se concibió originalmente) sino fuera de él, y poder utilizarlo en sistemas de escritorio o servidores web.

Este gestor de paquetes (muy similar al concepto de `apt-get` en GNU/Linux), nos permitirá instalar de forma muy sencilla y automática paquetes Javascript (tanto de Node como Javascript para el navegador) para poder utilizarlos y mantenerlos en los proyectos o sistemas que utilicemos.

Para comenzar, necesitaremos instalar **NodeJS** en su versión **LTS** (recomendada) o en su última versión (experimental) si quieres tener las últimas novedades. Al instalar Node, se instalará automáticamente su gestor de paquetes y algunas otras utilidades interesantes que necesitaremos.

En primer lugar, vamos a comprobar si tenemos **NodeJS/NPM** instalado y que versión tenemos:

```
# Muestra la versión de Node
$ node --version

# Muestra la versión de NPM
$ npm --version
```

En el caso de que no tengamos node instalado en nuestro sistema, se nos mostrará un mensaje de error como *node: command not found* o similar.

Si tenemos instalado **NodeJS** y **NPM** podremos **instalar paquetes NPM** en nuestros proyectos y/o sistema:

- **A nivel de proyecto:** Probablemente la modalidad más utilizada es la de usar NPM como un gestor de dependencias de un proyecto, esto es, un sistema con el que controlamos que paquetes o librerías Javascript están instalados (y que versión), de modo que quedan asociados al proyecto en sí. Esto facilita que si un usuario diferente se descarga el proyecto, pueda gestionarlo fácil y rápidamente (instalar paquetes, actualizarlos, etc...)
- **A nivel global:** Existen algunas situaciones específicas, en las que los paquetes son realmente utilidades que no se utilizan en proyectos, muy común en aplicaciones de **línea de comandos** (CLI como podría ser *gulp-cli*) que usamos desde terminal. En esta modalidad, los paquetes se instalan a nivel del sistema (no en la carpeta del proyecto), por lo que están disponibles siempre que el usuario quiera utilizarlos, sin necesidad de tenerlo en cada proyecto.

Existen alternativas a NPM:

- [PNPM](#)
- [Ultra](#)
- [Yarn](#)

Instalación de Node/NPM mediante NVM (Node Version Manager)

La ventaja de utilizar **NVM** para instalar y gestionar **NodeJS** es que podremos instalar diferentes versiones en un mismo sistema y/o cambiar de una versión a otra simplemente ejecutando un comando.

Además, al utilizar esta herramienta se evitan muchos problemas de permisos, sobretodo en sistemas GNU/Linux.

Instalación de NVM

NVM tiene versiones para GNU/Linux y para Windows. Descargamos el fichero `.msi` de la [web](#) y se instala como cualquier otro ejecutable de Windows.

NOTA: Según se puede ver en la web de [NVM-Windows](#), su autor está trabajando en el sucesor de esta herramienta llamada ***Runtime***.

Comandos de NVM

En este punto, tenemos instalado **NVM**, pero aún no tenemos instalado **NodeJS**:

```
# Ayuda
$ nvm --help

# Listar las versiones de NodeJS en el sistema
$ nvm list

# Mostrar las versiones de NodeJS disponibles para descargar e instalar
$ nvm list available

# Instalar una determinada versión de NodeJS
$ nvm install <version>

# Mostrar la versión de NodeJS que se está utilizando
$ nvm current

# Cambiar de versión a otra versión instalada
$ nvm use <version>
```

Archivos .nvmrc

Una forma muy efectiva de mantener versiones consistentes de Node.js dentro de un proyecto es utilizando archivos `.nvmrc`. Estos sencillos archivos de texto, colocados estratégicamente en la raíz del directorio del proyecto, **especifican la versión exacta de Node.js que se requiere**.

Para crear este archivo, navegamos hasta el directorio raíz del proyecto y ejecutamos este comando en el terminal para crear el fichero y guardar la versión requerida de Node.js (en el ejemplo, la versión 18.16.0):

```
echo 18.16.10 > .nvmrc
```

Una vez que el archivo `.nvmrc` esté en su lugar, se puede cambiar fácilmente a la versión de Node.js especificada usando el comando `nvm use`. NVM detectará automáticamente el archivo `.nvmrc` y cambiará a la versión especificada:

```
$ nvm use

# Found '/Users/**/.nvmrc' with version <v18.16.10>
# Now using node v18.16.10 (npm v10.8.3)
```

Aunque los archivos `.nvmrc` son increíblemente efectivos, también es recomendable declarar explícitamente la versión requerida de Node.js en el archivo **package.json**. Esto proporciona una capa adicional de claridad y puede ser utilizado por diversas herramientas y servicios que dependen del archivo **package.json** para obtener información esencial.

```
{
  "name": "your-project-name",
  "version": "1.0.0",
  "engines": {
```

```
"node": "18.16.0"
}
}
```

Uso de NPM

Instalaciones globales

Para instalar paquetes con NPM sin asociarlos a un proyecto deberemos utilizar el flag `--global` o `-g`. Esto indicará a **NPM** que el paquete debe ser instalado en el sistema y estará disponible para los usuarios del sistema, sin necesidad de tenerlo como dependencia del proyecto.

La recomendación es instalar los paquetes de forma global cuando:

1. Se trate de un paquete de línea de comandos
2. El paquete es para uso individual y no requiere estar ligado a un proyecto.
3. Si el paquete se utiliza en un proyecto, cada usuario que descargue el proyecto debe instalarlo de forma manual ya al instalar las dependencias del proyecto, no se instalan las globales.

```
# Instalar un paquete de forma global
$ npm install -g firebase-tools@latest
$ npm install --global gulp-cli@latest

# Ver los paquetes instalados de forma global
$ npm list -g --depth=0

# Desinstalar un paquete global
$ npm uninstall -g gulp-cli@latest
```

Administración de proyectos

NPM (*Node Package Manager*) nos permite, entre otras cosas, gestionar las dependencias de un proyecto de modo que no tenemos que hacerlo manualmente, sino de una forma relativamente automática y controlada, acelerando el desarrollo y reduciendo el tiempo necesario para mantener nuestros proyectos.

Dentro de la carpeta del proyecto ejecutamos el comando `npm init -y` para inicializar el proyecto, lo que creará el fichero `package.json`.

```
# Iniciar el control de versiones en el proyecto
$ git init

# Añadir la URL de Github (u otro servicio)
$ git remote add origin <URL>

# Inicializar el proyecto con NPM, que creará el fichero 'package.json'
$ npm init -y
```

NOTA: si hemos inicializado `git` en el proyecto y definido la URL, el fichero `package.json` tendrá las entradas **repository**, **bugs** y **homepage**.

Para trabajar en un proyecto, es necesario conocer bien el *scaffolding* (o estructura de carpetas). Un buen punto de partida podría ser:

```
- frontend-project/      # Carpeta raíz del proyecto
- .git/                  # Carpeta oculta de datos de git
- node_modules/          # Carpeta de paquetes de Node/NPM
- dist/                  # Carpeta de código generado
- src/                   # Carpeta de código fuente (código editable)
- assets/                # Carpeta de estáticos (imágenes, audio, etcétera...)
- js/                   # Carpeta de Javascript
-   - index.js
-   - css/               # Carpeta de CSS
-     - index.css
-   - index.html
- package.json           # Archivo del proyecto NPM
- package-lock.json      # Histórico de versiones de dependencias
- .gitignore             # Ficheros y carpetas a ignorar por git
```

El archivo `package-lock.json` es un archivo generado automáticamente cuando se instalan paquetes o dependencias en el proyecto. Su finalidad es mantener un historial de los paquetes instalados y optimizar la forma en que se generan las dependencias del proyecto y los contenidos de la carpeta `node_modules/`.

Este archivo debe **conservarse e incluso versionarse** para añadirlo al repositorio de control de versiones. Por tanto no debe añadirse al fichero `.gitignore`.

Instalación local en el proyecto

```
# Realiza una búsqueda de paquetes en NPM
$ npm search <paquete>

# Muestra información detallada del paquete
$ npm show <paquete>

# Instala el paquete en el proyecto como dependencia de desarrollo
$ npm install -D <paquete@latest>
$ npm install --save-dev <paquete@latest>

# Instala el paquete en el proyecto como dependencia de producción (por defecto)
$ npm install <paquete@latest>
$ npm install -P <paquete@latest>
$ npm install --save-prod <paquete@latest>

# Desinstalar un paquete, ya sea dependencia de desarrollo o de producción
$ npm uninstall -D <paquete@latest>
$ npm uninstall -P <paquete@latest>

# Ver las dependencias de desarrollo instalados de forma local
$ npm list -dev --depth=0

# Ver las dependencias de producción instalados de forma local
$ npm list -prod --depth=0
```

Las **dependencias de desarrollo** se guardarán en la sección `devDependencies` del fichero `package.json`. Son dependencias que únicamente estarán disponibles en la fase de desarrollo.

Las **dependencias de producción** se guardarán en la sección `dependencies` del fichero `package.json`. Son dependencias que se incluirán en la construcción del proyecto.

Si tenemos un paquete ya instalado en la modalidad de desarrollo y nos damos cuenta que realmente queremos tenerlo en la modalidad de producción (o viceversa), no es necesario desinstalar y volver a instalar el paquete.

Simplemente, escribimos el comando de instalación de la modalidad a la que queremos pasar (`npm install --save-dev <paquete@version>` o `npm install --save-prod <paquete@version>`) e internamente **NPM** se encargará de modificar el fichero `package.json` según sea necesario.

Una vez instalados los paquetes, tendremos nuestro proyecto listo para realizar *bare imports*, es decir, importaciones sin ruta (solo con el nombre del paquete), que node buscará en la carpeta `node_modules/`:

```
// Importamos La librería Howler de la carpeta node_modules/  
import { Howler, Howl } from "howler";  
  
const audio = new Howl({  
  src: ["/assets/audio.mp3"]  
});
```

Instalar versiones específicas

Si en algún momento necesitamos instalar una versión específica de un paquete, podemos hacerlo indicando la versión tras el nombre del paquete, separándola mediante el carácter `@`.

También podemos instalar determinadas versiones mediante su tag, como *latest*, *beta*, etcétera...:

```
# Muestra las versiones disponibles del paquete @vue/cli  
$ npm show @vue/cli versions  
  
# Muestra los tags disponibles  
$ npm show vite dist-tags  
  
# Instala la versión 4.4.6 de @vue/cli  
$ npm install -D @vue/cli@4.4.6  
  
# Instala la última versión disponible  
$ npm install -D @vue/cli@latest
```

Comando `npx`

Hasta ahora hemos hablado de instalar paquetes de línea de comandos de forma global, pero también es posible instalarlos localmente en un proyecto. Aquí es donde cobra especial importancia el comando `npx`.

Un problema común es que, si instalamos un paquete CLI (de línea de comandos) a nivel de proyecto, pero ya existe otra versión instalada globalmente en el sistema, puede que esta última tenga preferencia al ejecutarse. Esto ocurre porque el sistema busca primero en el PATH, donde suelen estar los paquetes globales.

Además, incluso si el paquete está instalado solo localmente, es posible que el sistema no lo encuentre directamente al ejecutarlo, ya que no está en el PATH global, sino en la carpeta del proyecto `node_modules/.bin`.

Para solventar esto, podemos utilizar `npx`, que nos permite ejecutar comandos CLI instalados localmente de forma transparente:

```
# Instalamos el paquete en el proyecto actual  
$ npm install -D gulp-cli@latest  
  
# Ejecutamos el comando  
$ npx gulp --version
```

El comando `npx` hace lo siguiente:

1) Busca el ejecutable en `node_modules/.bin`. Si lo encuentra, lo ejecuta desde ahí. 2) Si no lo encuentra localmente, busca una versión instalada globalmente. 3) Si tampoco existe globalmente, puede descargar temporalmente el paquete desde el registro de npm, ejecutarlo y luego eliminarlo (esta funcionalidad era común en versiones antiguas de `npx`, pero se ha ido limitando en versiones modernas).

NOTA: En versiones modernas de npm (v7 en adelante), se recomienda usar `npm exec`, especialmente si se quiere un comportamiento más controlado y predecible.

- <https://docs.npmjs.com/cli/v11/commands/npm-exec>
- <https://docs.npmjs.com/cli/v11/commands/npx>

Scripts NPM

NPM proporciona un sistema denominado **scripts de NPM**, que es algo así como una colección de tareas de línea de comandos.

La idea es incorporar en el fichero `package.json`, una colección de scripts que pueden ser abreviados con un nombre de tarea y ejecutados desde la carpeta raíz del proyecto. Suele ser común encontrar scripts para las tareas más comunes como **start**, **dev**, **serve**, **build**, **test** o **deploy**.

También es posible incluir los comandos de **build** y de **deploy** en una misma tarea, uniéndolos con `&&`, lo que en terminal significa: «si el comando anterior termina correctamente, ejecuta el siguiente», aunque esto puede complicar la legibilidad de los scripts.

Por ejemplo, podemos tener una tarea que elimine los ficheros anteriores de la carpeta `build\` y luego realiza la construcción:

```
"scripts": {
  "clean": "del-cli build/* dist/* .cache/*",
  "preview": "parcel build src/index.html -d build",
  "build": "del-cli build/* && parcel build src/index.html -d build"
}
```

```
# Ejecutar un script en NPM
$ npm run <nombre>
```

Scripts en paralelo

En caso de necesitar ejecutar scripts en paralelo y no tener que lidiar con la complejidad de herramientas como `Gulp` o `Grunt` podemos usar el paquete `npm-run-all`.

```
# Instalar La herramienta como dependencia de desarrollo
$ npm install --save-dev npm-run-all
```

Una vez instalado, si necesitamos que se ejecuten múltiples tareas que queremos realizar de forma independiente podemos agruparlas con un *namespace* utilizando los dos puntos `:` y luego ejecutar el grupo entero con un `$ npm-run-all`

`<namespace>:* :`

```
"scripts": {
  "build": "npm-run-all build:*",
  "build:html": "pug src/index.pug ...",
  "build:css": "postcss src/index.css ...",
  "build:js": "rollup src/index.js ..."
}
```

Por defecto, estas tareas se ejecutan **secuencialmente**. Es decir, hasta que la primera no termine, no se ejecutará la siguiente. Esto se puede modificar utilizando el flag `-p` (parallel):

```
"scripts": {
  "dev": "npm-run-all -p dev:*",
  "dev:html": "pug --watch src/index.pug ...",
  "dev:css": "postcss --watch src/index.css ...",
  "dev:js": "rollup --watch src/index.js ..."
}
```

Actualizar versiones locales y/o globales

El propio **NPM** tiene un comando para mostrar las dependencias no actualizadas, tanto a nivel de proyecto **local** como a nivel **global**:

```
# Mostrar las versiones no actualizadas a nivel local
$ npm outdated

# Mostrar las versiones no actualizadas a nivel global
$ npm -g outdated

# Actualiza las versiones manteniendo la política de semver
$ npm update

# Actualiza el número de versión en el package.json
$ npm update --save
```

Sin embargo podemos utilizar herramientas como [npm-check-updates](#) o [npm-check](#):

```
# Check the latest versions of all project dependencies
$ ncu

# Upgrade a project's package file. This will overwrite your package file.
$ ncu -u

# Check global packages
$ ncu -g
```

La actualización de dependencias tendrá en cuenta **la política de versionado semántico** de las dependencias.

NOTA: Una vez actualizadas las dependencias del proyecto se **deben instalar** con `npm install`.

Versionado semántico ('semver')

La forma más habitual de versionar nuestros paquetes es usando versionado semántico, muy frecuentemente abreviado como **'semver'**. Este tipo de versionado se basa en unas pautas concretas en el número de versión de los paquetes o proyectos. Dado un número de versión de un proyecto, este número está formado por 3 partes, por ejemplo: 2.0.4.

- **2** es el **major version**. Se trata del número de versión mayor y se avanza uno más si las novedades del paquete, librería o proyecto van a producir **cambios importantes** (*'breaking changes'*) que no son compatibles con versiones anteriores. Si avanzáramos, pasaríamos a 3.0.0. Si tu paquete no está siendo usado en producción, puedes empezar con la versión 0.1.0 en lugar de la 1.0.0.
- **0** es el **minor version**. Se trata del número de versión menor y se avanza uno más si las novedades del paquete, librería o proyecto introducen **nuevas funcionalidades** que son compatibles con versiones anteriores. Si avanzáramos, pasaríamos a 2.1.0.
- **4** es el **patch version**. Se trata del número utilizado para **parches** que corrigen problemas, bugs o defectos en funcionalidades del código actual y son compatibles con versiones anteriores. Si avanzáramos, pasaríamos a 2.0.5.

Política de actualizaciones

Si echamos un vistazo a las versiones de las dependencias de nuestro proyecto en el fichero `package.json`, veremos que tienen uno de los siguientes formatos:

- 1.2.4 -> Instala siempre la versión indicada 1.2.4
- ^1.2.4 -> Instalará la versión menor más alta, sin llegar a pasar a la versión 2.x ni superiores
- ~1.2.4 -> Instalará sólo los parches, sin llegar a pasar a la versión 1.3 ni superiores

Por defecto siempre se aplica la opción con el **circunflejo (^)**. De esta forma, sabemos que versión se instaló por primera vez, pero **NPM** buscará la versión más alta disponible, teniendo en cuenta sólo parches y actualizaciones menores, pero **nunca instalando actualizaciones mayores**.

También es posible utilizar el carácter `x` o `*` para indicar comodines en la versión. Por ejemplo, la versión `1.2.x` sería equivalente a `~1.2.0`, la versión `1.x` sería equivalente a `^1.0` y la versión `*` o `x` sería equivalente a cualquier versión.

Otros comandos de NPM

```
# Verificar el estado de La instalación de npm
$ npm doctor

# Auditar la seguridad de Los paquetes instalados localmente en el proyecto
$ npm audit

# Auditar e intentar corregir Los problemas detectados
$ npm audit fix

# Accede a La homepage de documentación del paquete "react"
$ npm docs react
$ npm home react

# Accede al repositorio del paquete "react"
$ npm repo react

# Accede a La página de issues del paquete "react"
$ npm issues react
$ npm bugs react

# Revisa La carpeta 'node_modules' en busca de paquetes que ya no se utilicen
$ npm prune

# Unificar paquetes redundantes (NPM lo hace automáticamente cuando se instalan)
$ npm dedupe
```

Enlaces

NPM

- <https://www.npmjs.com/>
- <https://docs.npmjs.com/>
- <https://www.npmjs.com/package/npx>
- <https://www.npmjs.com/package/npm-check-updates>
- <https://lenguajejs.com/npm/>
- <https://www.freecodecamp.org/espanol/news/node-js-npm-tutorial/>
- <https://semver.org/lang/es/>
- <https://dev.to/feliperocha93/ensuring-consistent-nodejs-versions-across-your-team-364d>
- [Node Version Manager](#)

- [NVM for Windows](#)
- [Volta](#)

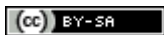
Node.js

- <https://nodejs.org/es>
- <https://openjsf.org/>
- <https://libuv.org/>
- <https://v8.dev/>
- <https://semver.org/lang/es/>

Misc

- <https://developer.chrome.com/docs/devtools>

Licencia



Esta obra está bajo una [licencia de Creative Commons Reconocimiento-Compartir Igual 4.0 Internacional](#).