Tema 4 - Angular

Introducción a Node Package Manager (NPM)

Desarrollo web en entorno cliente
IES Pere Maria Orts I Bosch

Índice

Introducción

Podemos decir que NPM (Node Package Manager) es una herramienta que facilita reutilizar código de otros programadores en nuestros proyectos. Este código está distribuido en paquetes (packages) o módulos. Podemos encontrar algunas de las librerías más populares, frameworks y herramientas como jQuery, Angular, Express, JSHint, ESLint, Sass, Browserify, Bootstrap, Cordova, Ionic, entre otras.

Además, NPM nos proporciona un completo sistema de automatización de tareas basado en scripts. Para la mayoría de los casos, NPM es suficiente, siendo innecesario el uso de otras herramientas como Gulp, Grunt o Bower, aunque muchos desarrolladores las combinan en sus proyectos, pero con la desventaja de que necesitas aprender cómo funcionan 2, 3 o 4 herramientas diferentes.

Instalación de NPM

Para instalar NPM, vamos a necesitar instalar primero Node.js (NPM es parte de Node.js). Node permite la ejecución de aplicaciones JavaScript en el servidor usando el motor v8 (Chrome), permitiendo la creación de aplicaciones tanto de cliente como de servidor con un único lenguaje → JavaScript.

Podemos consultar las instrucciones de cómo instalarlo en Windows y Mac aquí. Si usamos Linux, hay varias distribuciones que incluyen Node en sus repositorios. Si queremos instalar la última versión disponible (añadiendo un repositorio externo), podemos seguir las instrucciones que hay aquí. La versión recomendable para instalar es la última 10.x, porque es la actual versión LTS, con soporte desde octubre de 2018 hasta abril de 2021.

Una vez que tengamos Node instalado, podemos comprobar si lo tenemos activado desde la línea de comando ejecutando:

```
arturo@arturo-desktop:~$ node -v
v10.16.3
arturo@arturo-desktop:~$ npm -v
6.11.3
```

Además, podemos actualizar NPM a la última versión disponible utilizando el comando npm install npm -g (en Linux ejecútalo como root o con sudo).

Paquetes NPM

Un paquete (librería, framework, tool, ...), es un directorio que contiene varios archivos, incluyendo un archivo package.json, que contiene información sobre el autor, versión, otros paquetes de los que a su vez depende este paquete, etc. En esta sección vamos a ver cómo instalar, actualizar, eliminar, etc, paquetes en nuestro proyecto (o de forma global en nuestro sistema).

Podemos consultar la ayuda de npm mediante los comandos:

- npm -h → Muestra la ayuda rápida y una lista con los comandos típicos de npm
- npm comando -h → Muestra una ayuda rápida con información específica sobre el comando npm (npm install -h).
- npm help comando → Se abre una página en el navegador o el man de Linux, mostrando ayuda sobre un determinado comando
- npm help-search palabras → Nos devuelve una lista de ayuda de aquellos que contiene la palabra(s) buscada(s) (separadas por espacios)

Creando nuestro package.json

Hay dos tipos de proyectos que podemos crear, una página o aplicación web que ejecutarán los usuarios, o una librería/herramienta para ser incluida en proyectos de otros desarrolladores. Ambos tendrán dependencias de otros paquetes y también necesitan ser probadas, *minificadas*, etc..

Para crear un archivo package.json en nuestro proyecto, simplemente nos moveremos al directorio principal de nuestro proyecto (no hace falta que esté vacío) y ejecutamos npm init. Nos preguntará algunas cosas sobre nuestro proyecto y creará el archivo package.json basado en eso. Podemos dejar muchos de los valores por defecto o vacíos presionando enter. Justo antes de crear el archivo, nos mostrará el contenido que tendrá.

```
{
  "name": "project",
  "version": "1.0.0",
  "description": "A project created to learn NPM",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Arturo",
  "license": "ISC"
}
```

Instalando paquetes

El comando para instalar un paquete en nuestro proyecto es npm install

nombre-paquete. Si ejecutamos este comando por primera vez, veremos que creará un directorio llamado node_modules dentro del directorio de proyecto. Este directorio contendrá todos los paquetes instalados mediante npm (incluyendo a su vez las dependencias de dichos paquetes). Vamos a probar con jQuery.

```
arturo@arturo-Lenovo:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ npm install jquery
project@1.0.0 /home/arturo/Dropbox/Public/2016-2017/DAWEC/pruebas/project
└── jquery@3.1.0

npm WARN project@1.0.0 No repository field.
arturo@arturo-Lenovo:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ ls node_modules/
jquery
```

Como vemos, instalará jQuery (la última versión) dentro del directorio node_modules (el archivo que enlazaremos en nuestro proyecto estará en jquery/dist/).

Incluyendo las dependencias en package.json

Para incluir una dependencia en nuestro archivo package.json, usamos la opción --save o -S junto al comando de instalación (i → alias de *install*). Nota: desde la versión 5 de NPM se puede omitir esta opción ya que lo hace por defecto.

```
npm i jquery -S
```

```
{
    "name": "project",
    "version": "1.0.0",
    "description": "Proyecto creado para aprender NPM",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "Arturo",
    "license": "ISC",
    "dependencies": {
        "jquery": "^3.1.0"
    }
}
```

Esto nos asegura que NPM instalará o actualizará a la última versión "3.x.x" de la librería jQuery en nuestro proyecto. Por defecto, será considerado como una dependencia de producción (nuestra aplicación la necesita para ejecutarse). Hay otros paquetes que no se necesitan al ejecutar la aplicación, sino por otros motivos, como pruebas (ejemplo: karma), o minificar/ofuscar código (ejemplo: uglify). Estas se conocen como dependencias de desarrollo y normalmente no están incluidas (se pueden desinstalar) cuando nuestra aplicación está en producción.

Para incluir un paquete en nuestra lista de dependencias de desarrollo usaremos la opción –save-dev o -D. Por ejemplo:

```
npm i uglify --save-dev
```

```
{
    "name": "project",
```

```
    "version": "1.0.0",
    "description": "Proyecto creado para aprender NPM",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "author": "Arturo",
    "license": "ISC",
    "dependencies": {
        "jquery": "^3.1.0"
    },
    "devDependencies": {
        "uglify": "^0.1.5"
    }
}
```

Instalando las dependencias existentes del proyecto

Si descargas un proyecto existente con un archivo package.json (o clonamos un repositorio GIT), normalmente no tendrá las dependencias instaladas (ocupan mucho). En resumen, el directorio node_modules se añade a .gitignore para ser ignorado por GIT. Para volver a instalar las dependencias tanto para producción como para desarrollo que aparecen en package.json ejecutamos npm install (o npm i).

Podemos instalar solo las dependencias de producción o solo las dependencias de desarrollo usando npm install --only=prod, o npm install --only=dev.

Instalando paquetes globales

Hay herramientas de JavaScript como handlebars, gulp, grunt, etc. que pueden ser instaladas como globales en el sistema operativo, y de esta forma, usarlas en cualquier directorio o proyecto.

Para instalar una dependencia de forma global, lo que haremos será incluir en el comando de la instalación la opción -g al final del comando. Hay que tener en cuenta que debemos ser administradores (root o sudo en Linux). Por ejemplo:

npm i handlebars -g → Instala el comando handlebars de forma global

```
arturo@arturo-sobremesa:~$ handlebars -v
4.0.5
```

Listando los paquetes instalados

Para poder ver los paquetes instalados en el directorio npm_modules escribiremos npm list. Nos mostrará mediante una vista de árbol, los paquetes que hay instalados y las dependencias que tiene cada uno de ellos. Para ver sólo los paquetes que hemos instalado nosotros (sin las dependencias de estos paquetes), ejecutamos npm list --depth=0.

```
arturo@arturo-sobremesa:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ npm list --depth=0
project@1.0.0 /otros/Dropbox/Public/2016-2017/DAWEC/pruebas/project
├── jquery@3.1.0
└── uglify@0.1.5
```

Si queremos mostrar un listado de los paquetes globales del sistema,

necesitamos añadir la opción --global=true al comando.

```
arturo@arturo-sobremesa:~$ npm list --global=true --depth 0
/usr/lib
├── docker@0.2.14
├── handlebars@4.0.5
└── npm@3.10.6
```

Otra opción bastante útil es incluir --dev=true (lista sólo los paquetes en desarrollo), --prod=true (solo los paquetes en producción), o --long=true (muestra una descripción de cada uno).

```
arturo@arturo-sobremesa:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ npm list --long=true --
depth=0
project@1.0.0
| /otros/Dropbox/Public/2016-2017/DAWEC/pruebas/project
| A project created to learn NPM
├── jquery@3.1.0
|   JavaScript library for DOM operations
|   git+https://github.com/jquery/jquery.git
|   https://jquery.com
└── uglify@0.1.5
    A simple tool to uglify javascript & css files
    git://github.com/nanjingboy/uglify.git
    https://github.com/nanjingboy/uglify
```

Instalando una versión específica de un paquete

Por defecto, cuando instalamos un paquete, se instalará la última versión estable. Pero a veces, nuestro proyecto necesita incluir una versión anterior (por ejemplo, si queremos soportar IExplorer 8, necesitamos la versión de jQuery 1.x).

Las versiones de los paquetes normalmente tienen tres números X.Y.Z. La mayoría de paquetes siguen las reglas SemVer (Semantic Versioning) para aplicar la numeración. Son estas:

- Cuando la Z es incrementada, significa que un bug o problema ha sido solucionado, pero no se añade ninguna funcionalidad nueva.
- Cuando la Y es incrementada, significa que nuevas funcionalidades han sido añadidas, pero que esto no afecta al código de las versiones previas (siempre que se mantenga la X). Por ejemplo, una aplicación hecha con AngularJS 1.2 debería seguir funcionando con AngularJS 1.5 (pero al revés no tiene por qué).
- El número X se incrementará sólo si han habido suficientes cambios de forma que no se garantiza que las versiones anteriores (con la X menor) funcionen o sean compatibles con la nueva. Por tanto debemos tener cuidado cuando hagamos una migración y vigilar los cambios que se han producido en la librería o framework. Por ejemplo, la versión jQuery 2.x no soporta IE8 ni versiones anteriores, sin embargo la versión 1.x sí.

Si queremos instalar una versión específica de una librería en lugar de la última versión, lo que debemos hacer es escribir paquete@x.y.z en lugar de sólo el nombre.

npm i jquery@"1.12.3" → Instalará esta versión de jQuery que es todavía compatible con IE8. Si añadimos la opción --save, nos la guardará en el archivo package.json y nunca será actualizada a menos que cambiemos nosotros la versión.

También podemos especificar que queremos instalar una versión previa a x.y.z con: pakage@"<x.y.z".

Otras posibilidades (ejemplos):

- "*" ó "x" → Instala la última versión de un paquete (cuidado con los cambios que pueda tener esa versión y cómo nos puede afectar...).
- "3" ó "3.x", ó "3.x.x" → Esto instala la última versión de un paquete siempre que sea 3.x.x (no se actualizará a la versión 4.x.x).
- "^3.3.5" → Instalará la última versión 3 (3.x.x) de un paquete, pero como mínimo deberá ser la versión 3.3.5 (el carácter ^ significa que sólo el primer número, 3, debe respetarse). Este es el comportamiento por defecto de NPM
- "3.3" ó "3.3.x" → Instalará la última versión 3.3.x de un paquete (nunca se actualizará a 3.4.x o posterior).
- "~3.3.5" → Instalará la última versión de 3.3 (no actualizará a la 3.4 o superior) pero como mínimo deberá ser la versión 3.3.5 (el carácter ~ significa que los 2 primeros números, 3.3, deben respetarse). Equivale a "3.3.x".

Actualización de paquetes

Para actualizar todas las dependencias de un proyecto debemos ejecutar npm update (mirará en nuestro archivo package.json para ver a qué versiones se permite actualizar). Para actualizar un paquete en concreto utilizaremos npm update paquete. Podemos usar --prod o --dev para actualizar sólo paquetes en producción o en desarrollo. O -g para actualizar un paquete global.

Eliminando paquetes

Para eliminar un paquete de nuestro proyecto teclearemos npm uninstall package. Si añadimos --save, nos eliminará también sus dependencias del proyecto (si no necesitamos ese paquete nunca más). En lugar de uninstall podemos usar remove, rm, un, r o unlink para hacer lo mismo. Para desinstalar un paquete global usaremos la opción -g.

npm r jquery --save → Elimina JQuery (y sus dependencias) del proyecto y también del archivo package.json.

npm prune --production → Elimina las dependencias de desarrollo dejando sólo las dependencias de producción.

Automatizandotareas con scripts en NPM

En nuestro archivo package.json, además de administrar las dependencias, podemos crear algunos scripts útiles para nuestro proyecto (ejecutar un servidor web, testear nuestra aplicación, minificar, etc.). Estos scripts son comandos y todos tienen un nombre que les identifica. Debemos poner estos scripts dentro de la sección "script" (es un array). La sintaxis del script será: "nombre-script": "Comando a ejecutar". Para ejecutar un script usaremos: npm run nombre-script.

```
"scripts": {
    "hello": "echo 'hello'"
}
```

```
arturo@arturo-Lenovo:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ npm run hello

> project@1.0.0 hello /home/arturo/Dropbox/Public/2016-2017/DAWEC/pruebas/project
> echo 'hello'

hello
```

Scripts de inicio

Hay algunos scripts como start, stop o test que se pueden ejecutar sin la necesidad de escribir run. El script *start* normalmente se usa para ejecutar un servidor web, y por tanto probar nuestra aplicación. A veces, se usa para ejecutar un compilador de TypeScript o de SASS, por ejemplo.

En nuestro ejemplo, el script *start* abrirá el navegador chrome con una URL donde tendremos nuestro servidor web ejecutándose, de forma que podremos probar nuestra aplicación:

```
"scripts": {
    "start": "google-chrome http://localhost/project"
}
```

Ejecutaríamos este script escribiendo npm start.

Script de test

En una aplicación seria, querremos probar nuestro código antes de publicarlo, por tanto deberíamos crear un script (o más) que ejecute los tests.

Vamos a hacer un ejemplo para depurar un archivo JavaScript con ESLint. Esta herramienta nos permite comprobar nuestro código (errores del código, buenas prácticas, etc.), pero no realizar test de unidad u otro tipo de tests (mejor usar otras herramientas como mocha, jasmine, karma,...).

Primero vamos a instalar de forma global ESLint ejecutando npm i eslint -g. Una vez instalado ejecutamos el comando: eslint --init dentro del directorio de

proyecto. Nos hará una serie de preguntas sobre la configuración de la herramienta:

```
arturo@arturo-Lenovo:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ eslint --init
? How would you like to configure ESLint? Answer questions about your style
? Are you using ECMAScript 6 features? No
? Where will your code run? Browser
? Do you use CommonJS? No
? Do you use JSX? No
? What style of indentation do you use? Tabs
? What quotes do you use for strings? Double
? What line endings do you use? Unix
? Do you require semicolons? Yes
? What format do you want your config file to be in? JSON
Successfully created .eslintrc.json file in /home/arturo/Dropbox/Public/2016-2017/DAWEC/pruebas/project
```

Se creará un archivo de configuración con las opciones elegidas (.eslintrc.json en mi caso, en Linux). Podemos cambiar algunas de las reglas, por ejemplo en lugar de mostrar un error, que muestre un warning (cambiando "error" por "warn"):

```
{
    "env": {
        "browser": true
    },
    "extends": "eslint:recommended",
    "rules": {
        "indent": ["error",  "tab"],
        "linebreak-style": ["error",  "unix" ],
        "quotes": ["warn", "double"],
        "semi": [error", "always"]
    }
}
```

Ahora, vamos a ejecutar el test sobre algún archivo JavaScript:

```
'strict mode';

function printNum() {
        num = 34; // Indentación con tabulador
    return num * 2; // Espacios en lugar de tabulado
}

alert('Hello World!');
printNum() // He olvidado poner el punto y coma ";"
```

```
arturo@arturo-Lenovo:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ eslint main.js

/home/arturo/Dropbox/Public/2016-2017/DAWEC/pruebas/project/main.js
  1:1   warning  Strings must use doublequote                        quotes
  4:2   error    'num' is not defined                                no-undef
  5:5   error    Expected indentation of 1 tab character but found 0 indent
  5:12  error    'num' is not defined                                no-undef
  8:7   warning  Strings must use doublequote                        quotes
  9:11  error    Missing semicolon                                   semi

✖ 6 problems (4 errors, 2 warnings)
```

Ahora podemos poner el comando en nuestro package.json, para que cuando ejecutemos npm test (o npm tst), ejecute eslint sobre ese fichero:

```
"scripts": {
    "start": "google-chrome http://localhost/project",
```

```
    "test": "eslint main.js"
}
```

Tarea previa o posterior al script (hooks)

Algunas tareas requieren realizar algunos pasos previos como por ejemplo minificar nuestro código JavaScript o CSS antes de ejecutar la aplicación (start script). Para ejecutar más de un comando podemos enlazarlos usando && (and). Estos comandos se ejecutarán en orden (excepto si uno de ellos falla).

```
"scripts": {
    "start": "google-chrome http://localhost/project",
    "test": "echo 'We are going to make some tests' && eslint main.js && echo
'Test successful!'",
}
```

Sin embargo, es más limpio si usamos los *hooks* de NPM (prefijos) pre y post. Cuando creas un script que se ejecute junto a otro y lo quieras ejecutar antes usarás (pre), o después (post) seguido del nombre del script principal. En este ejemplo, crearemos un script pretest y un posttest.

```
"scripts": {
    "start": "google-chrome http://localhost/project",
    "test": "eslint main.js",
    "pretest": "echo 'We are going to make some tests'",
    "posttest": "echo 'Test successful!'"
}
```

Ahora, cuando ejecutemos test, se ejecutará primero pretest, después test, y finalmente posttest en este orden. Si un script falla, el siguiente no será ejecutado.

```
arturo@arturo-Lenovo:~/Dropbox/Public/2016-2017/DAWEC/pruebas/project$ npm test -s
We are going to make some tests

/home/arturo/Dropbox/Public/2016-2017/DAWEC/pruebas/project/main.js
  1:1  warning  Strings must use doublequote  quotes

✖ 1 problem (0 errors, 1 warning)

Test successful!
```

Ejemplo: Minificar con uglify

Uglify es una herramienta que minifica (comprime) archivos JavaScript, haciéndolos más ligeros, lo que hace que nuestras páginas carguen más rápido, y más complejos de entender o leer. En nuestro proyecto, uglify se instalaría como una dependencia de desarrollo (npm i uglify-js -D).

Ahora, sólo necesitamos añadir una línea como esta en la parte de los script, lo cual comprimirá y unificará todos los archivos JS del directorio js/ en un único archivo: bundle.js (que incluiremos en nuestro HTML):

```
"build": "uglifyjs -mc -o bundle.js js/*.js"
```

Si ejecutamos npm run build, nos generará bundle.js con el código JS minificado dentro:

```
"strict mode";function printNum(){var r=34;return 2*r}alert("Hello
World!"),printNum();
```

Ejemplo: Ejecutar un script cuando algunos archivos cambian

Hay una herramienta que puede utilizarse cuando alguno de los archivos que tenemos en un directorio cambia. Esta herramienta se llama watch, y podemos añadirla a nuestras dependencias del proyecto ejecutando npm i watch --save-dev.

Para usar esta herramienta, crearemos un script que ejecute: watch 'comando' directorio. Cuando se produzca un cambio en un archivo dentro del directorio especificado, el comando será ejecutado de forma automática.

```
"build": "uglifyjs -mc -o bundle.js js/*.js",
"build:watch": "watch 'npm run build' ./js" → Cambio detectado en js/
```

Ahora, si ejecutamos npm run build:watch, esta herramienta estará de forma continua vigilando y cuando un archivo del directorio js cambie, se ejecutará el script build. Para pararlo, pulsaremos Ctrl+c en la consola.

Ejemplo: Lanzando un servidor web y vigilando cambios de forma concurrente

Si ejecutamos un servidor web usando NPM (lite-server por ejemplo), no podremos ejecutar ninguna otra tarea hasta que esta acabe (cerrada con Ctrl+c por ejemplo). Hay un paquete en NPM que nos permite ejecutar más de una tarea de forma concurrente y se llama concurrently.

Primero instalaremos estos dos dependencias en modo desarrollo (npm i lite-server -D) (npm i concurrently -D). Creamos una tarea que ejecutará a su vez dos tareas a la vez, pasándole los comandos como parámetro (entre comillas).

```
"start": "concurrently \"npm run build:watch\" \"npm run serve\"",
"serve": "lite-server",
"build": "uglifyjs -mc -o bundle.js js/*.js",
"build:watch": "watch 'npm run build' ./js"

"devDependencies": {
  "concurrently": "^2.2.0",
  "lite-server": "^2.2.2",
  "uglify-js": "^2.7.10",
  "watch": "^0.19.2"
}
```

lite-server carga por defecto el archivo con nombre index.html como punto de partida.