Usando módulos de terceros

Además de poder utilizar módulos incorporados en Node.js, o descomponer nuestra aplicación en diversos módulos conectados, Node.js es un framework con una gran comunidad de desarrollo detrás, que ya ha desarrollado una gran variedad de módulos o librerías que podemos utilizar libremente en nuestros proyectos, con el objetivo (siempre presente) de no tener que "reinventar la rueda", y además, utilizar un código que ha sido ampliamente probado y utilizado por otros desarrolladores.

1. El gestor de paquetes NPM

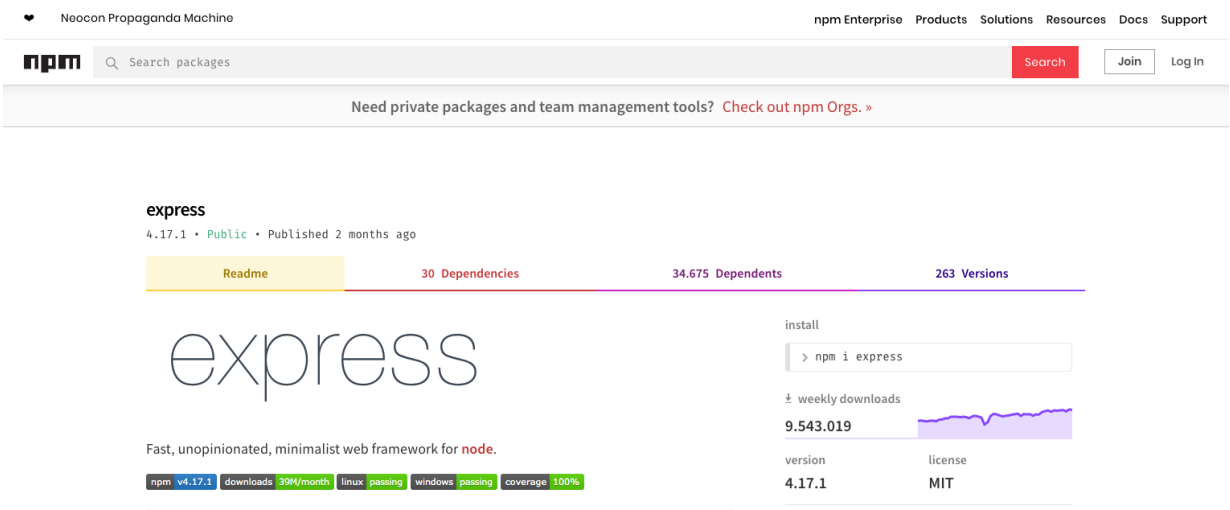
npm (*Node Package Manager*) es un gestor de paquetes para Javascript, y se instala automáticamente al instalar Node.js. Podemos comprobar que lo tenemos instalado, y qué versión concreta tenemos, mediante el comando:

```
npm -v
```

aunque también nos servirá el comando `npm --version`.

Inicialmente, `npm` se pensó como un gestor para poder instalar módulos en las aplicaciones Node, pero se ha convertido en mucho más que eso, y a través de él podemos también descargar e instalar en nuestras aplicaciones JavaScript (cliente o servidor) otros módulos o librerías que no tienen que ver con Node, como por ejemplo *Bootstrap* o *jQuery*. Así que actualmente es un enorme ecosistema de librerías open-source, que nos permite centrarnos en las necesidades específicas de nuestra aplicación, sin tener que "reinventar la rueda" cada vez que necesitemos una funcionalidad que ya han hecho otros antes.

El registro de librerías o módulos gestionado por NPM está en la web npmjs.com. Podemos consultar información sobre alguna librería en particular, consultar estadísticas de cuánta gente se la descarga, e incluso proporcionar nosotros nuestras propias librerías si queremos. Por ejemplo, esta es la ficha de la librería *express*, que emplearemos dentro de unas sesiones:



La opción más habitual de uso de *npm* es instalar módulos o paquetes en un proyecto concreto, de forma que cada proyecto tenga sus propios módulos. Sin embargo, en algunas ocasiones también nos puede interesar (y es posible) instalar algún módulo de forma global al sistema. Veremos cómo hacer estas dos operaciones.

2. Instalar módulos locales a un proyecto

En este apartado veremos cómo instalar módulos de terceros de forma local a un proyecto concreto. Haremos pruebas dentro de un proyecto llamado "*PruebaNPM*" en nuestra carpeta de "*ProyectosNode/Pruebas*", cuya carpeta podemos crear ya.

2.1. El archivo "package.json"

La configuración básica de los proyectos Node se almacena en un archivo JSON llamado `package.json`. Este archivo se puede crear directamente desde línea de comandos, utilizando una de estas dos opciones (debemos ejecutarla en la carpeta de nuestro proyecto Node):

- `npm init --yes`, que creará un archivo con unos valores por defecto, como éste que se muestra a continuación:

```
{
    "name": "PruebaNPM",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    "scripts": {
        "test": "echo \"Error: no test specified\" && exit 1"
    },
    "keywords": [],
    "author": "",
    "license": "ISC"
}
```

- `npm init`, que iniciará un asistente en el terminal para que demos valor a cada atributo de la configuración. Lo más típico es rellenar el nombre del proyecto (toma como valor por defecto el nombre de la carpeta donde está), la versión, el autor y poco más. Muchas opciones tienen valores por defecto puestos entre paréntesis, por lo que si pulsamos *Intro* se asignará dicho valor sin más.

Al final de todo el proceso, tendremos el archivo en la carpeta de nuestro proyecto. En él añadiremos después (de forma manual o automática) los módulos que necesitemos, y las versiones de los mismos, como explicaremos a continuación.

> **NOTA**: al generar el archivo `package.json`, podemos observar que el nombre de programa principal (*entry point*) que se asigna por defecto a la aplicación Node es *index.js*. Es habitual que el fichero principal de una aplicación Node se llame así, o también *app.js*, como veremos en posteriores ejemplos, aunque no es obligatorio llamarlos así.

2.2. Añadir módulos al proyecto y utilizarlos

Para instalar un módulo externo en un proyecto determinado, debemos abrir un terminal y situarnos en la carpeta del proyecto. Después, escribimos el siguiente comando:

```
npm install nombre_modulo
```

donde *nombre_modulo* será el nombre del módulo que queramos instalar. Podemos instalar también una versión específica del módulo añadiéndolo como sufijo con una arroba al nombre del módulo. Por ejemplo:

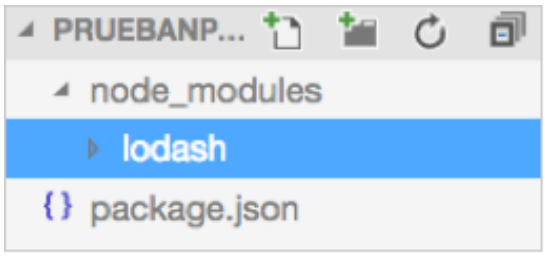
```
npm install nombre_modulo@1.1.0
```

Vamos a probar con un módulo sencillo y muy utilizado (tiene millones de descargas semanalmente), ya que contiene una serie de utilidades para facilitarnos el desarrollo de nuestros proyectos. Se trata del módulo `lodash`, que podéis consultar en la web citada anteriormente (aquí). Para instalarlo, escribimos lo siguiente:

```
npm install lodash
```

Algunas puntualizaciones antes de seguir:

- Tras ejecutar el comando anterior, se habrá añadido el nuevo módulo en una subcarpeta llamada `node_modules` dentro de nuestro proyecto.



- También tras ejecutar el comando anterior, se habrá añadido el módulo en una sección llamada *dependencies* dentro del archivo `package.json`.

```
{
    "name": "pruebanpm",
    ...
    "dependencies": {
        "lodash": "^4.17.21"
    }
}
```

- Al instalar cualquier nuevo módulo, se generará (o modificará) un archivo adicional llamado `package-lock.json`. Este archivo es un backup de cómo ha quedado el árbol de carpetas en "*node_modules*" con la nueva instalación, de forma que podamos volver atrás y dejar los módulos como estaban en cualquier paso previo. Es utilizado en repositorios git para estas restauraciones, precisamente. Nosotros no le haremos mucho caso en este curso.

Para poder utilizar el nuevo módulo, procederemos de la misma forma que para utilizar módulos predefinidos de Node: emplearemos la instrucción `require` con el nombre original del módulo. Por ejemplo, vamos a editar un archivo `index.js` en la carpeta "*PruebaNPM*" que venimos editando en estos últimos pasos, y añadimos este código que carga el módulo "lodash" y lo utiliza para eliminar un elemento de un vector:

```
const lodash = require('lodash');
console.log(lodash.difference([1, 2, 3], [1]));
```

> **NOTA:** si buscáis documentación o ejemplos de uso de esta librería en Internet, es habitual que el nombre de variable o constante donde se carga (en la línea `require`) sea un simple símbolo de subrayado (eso es lo que significa *low dash* en inglés), con lo que el ejemplo anterior quedaría así:

```
const _ = require('lodash');
console.log(_.difference([1, 2, 3], [1]));
```

Si ejecutamos este ejemplo desde el terminal, obtendremos lo siguiente:

```
node index.js
[ 2, 3 ]
```

2.3. Desinstalar un módulo

Para desinstalar un módulo (y eliminarlo del archivo `package.json`, si existe), escribimos el comando siguiente:

```
npm uninstall nombre_modulo
```

Tras este paso, el módulo se elimina tanto de la carpeta `node_modules` como de la sección *dependencies* del archivo `package.json`.

2.4. Orden de inclusión de los módulos

Hemos visto cómo incluir en una aplicación Node tres tipos de módulos:

- Módulos pertenecientes al núcleo de Node
- Módulos de nuestro propio proyecto, si está dividido en varios ficheros fuente
- Módulos hechos por terceros, descargados a través de *npm*

Aunque no hay una norma obligatoria a seguir al respecto, sí es habitual que, cuando nuestra aplicación necesita incluir módulos de diversos tipos (predefinidos de Node, de terceros y archivos propios), se haga con una estructura determinada.

Básicamente, lo que se hace es incluir primero los módulos de Node y los de terceros, y después (separados por un espacio del bloque anterior), los archivos propios de nuestro proyecto. Por ejemplo:

```
const fs = require('fs');
const _ = require('lodash');

const utilidades = require('./utilidades');
```

2.5. Algunas consideraciones sobre módulos de terceros

Hemos visto los pasos elementales para poder instalar, utilizar, y desinstalar (si es necesario) módulos de terceros localmente en nuestras aplicaciones. Pero hay algunos aspectos referentes a estos módulos, y la forma en que se instalan y distribuyen, que debes tener en cuenta.

Gestión de versiones

Desde que comenzamos a desarrollar una aplicación hasta que la finalizamos, o en mantenimientos posteriores, es posible que los módulos que la componen se hayan actualizado. Algunas de esas nuevas versiones pueden no ser compatibles con lo que en su día hicimos, o al contrario, hemos actualizado la aplicación y ya no nos sirven versiones demasiado antiguas de ciertos módulos.

Para poder determinar qué versiones o rangos de versiones son compatibles con nuestro proyecto, podemos utilizar la misma sección de "*dependencies*" del archivo `package.json`, con una nomenclatura determinada. Veamos algunos ejemplos utilizando el paquete "*lodash*" del caso anterior:

- `"lodash": "1.0.0"` indicaría que la aplicación sólo es compatible con la versión 1.0.0 de la librería
- `"lodash": "1.0.x"` indica que nos sirve cualquier versión 1.0
- `"lodash": "*"` indica que queremos tener siempre la última versión disponible del paquete. Si dejamos una cadena vacía "", se tiene el mismo efecto. No es una opción recomendable en algunos casos, al no poder controlar lo que contiene esa versión.
- `"lodash": ">= 1.0.2"` indica que nos sirve cualquier versión a partir de la 1.0.2.
- `"lodash": "< 1.0.9"` indica que sólo son compatibles las versiones de la librería hasta la 1.0.9 (sin contar esta última).
- `"lodash": ">= 1.0.2 < 1.0.9"` indica que sólo son compatibles las versiones de la librería desde la 1.0.2 (inclusive) hasta la 1.0.9 (sin contar esta última).
- `"lodash": "^1.1.2"` indica cualquier versión desde la 1.1.2 (inclusive) hasta el siguiente salto mayor de versión (2.0.0, en este caso, sin incluir este último).
- `"lodash": "~1.3.0"` indica cualquier versión entre la 1.3.0 (inclusive) y la siguiente versión menor (1.4.0, exclusive).

Existen otros modificadores también, pero con éstos podemos hacernos una idea de lo que podemos controlar. Una vez hayamos especificado los rangos de versiones compatibles de cada módulo, con el siguiente comando actualizamos los paquetes que se vean afectados por estas restricciones, dejando para cada uno una versión dentro del rango compatible indicado:

```
npm update
```

Añadir módulos a mano en "package.json"

También podríamos añadir a mano en el archivo "package.json" módulos que necesitemos instalar. Por ejemplo, así añadiríamos al ejemplo anterior la última versión del módulo "express":

```
{
    ...
    "dependencies": {
        "lodash": "^4.17.21",
        "express": "*"
    }
}
```

Para hacer efectiva la instalación de los módulos de este archivo, una vez añadidos, debemos ejecutar este comando en el terminal:

```
npm install
```

Automáticamente se añadirán los módulos que falten en la carpeta *node_modules* del proyecto.

Compartir nuestro proyecto

Si decidimos subir nuestro proyecto a algún repositorio en Internet como Github o similares, o dejar que alguien se lo descargue para modificarlo después, no es buena idea subir la carpeta "node_modules", ya que contiene código fuente hecho por terceras personas, probado en entornos reales y fiable, que no debería poderse modificar a la ligera. Además, la forma en que se estructura la carpeta "node_modules" depende de la versión de npm que cada uno tengamos instalada, y es posible que ocupe demasiado. De hecho, los propios módulos que descargamos pueden tener dependencias con otros módulos, que a su vez se descargarán en una subcarpeta interna.

Por lo tanto, lo recomendable es no compartir esta carpeta (no subirla al repositorio, o no dejarla a terceras personas), y no es ningún problema hacer eso, ya que gracias al archivo `package.json` siempre podemos (debemos) ejecutar el comando

```
npm install
```

y descargar todas las dependencias que en él están reflejadas. Dicho de otra forma, el archivo `package.json` contiene un resumen de todo lo externo que nuestro proyecto necesita, y que no es recomendable facilitar con el mismo.

3. Instalar módulos globales al sistema

Para cierto tipo de módulos, en especial aquellos que se ejecutan desde terminal como Grunt (un gestor y automatizador de tareas Javascript) o JSHint (un comprobador de sintaxis Javascript), puede ser interesante instalarlos de forma global, para poderlos usar dentro de cualquier proyecto.

La forma de hacer esto es similar a la instalación de un módulo en un proyecto concreto, añadiendo algún parámetro adicional, y con la diferencia de que, en este caso, no es necesario un archivo "package.json" para gestionar los módulos y dependencias, ya que no son módulos de un proyecto, sino del sistema. La sintaxis general del comando es:

```
npm install -g nombre_modulo
```

donde el flag `-g` hace referencia a que se quiere hacer una instalación global.

Es importante, además, tener presente que cualquier módulo instalado de forma global en el sistema no podrá importarse con `require` en una aplicación concreta (para hacerlo tendríamos que instalarlo también de forma local a dicha aplicación).

3.1. Ejemplo: nodemon

Veamos cómo funciona la instalación de módulos a nivel global con uno realmente útil: el módulo `nodemon`. Este módulo funciona a través del terminal, y nos sirve para monitorizar la ejecución de una aplicación Node, de forma que, ante cualquier cambio en la misma, automáticamente la reinicia y vuelve a ejecutarla por nosotros, evitándonos tener que escribir el comando `node` en el terminal de nuevo. Podéis consultar información sobre nodemon aquí.

Para instalar `nodemon` de forma global escribimos el siguiente comando (con permisos de administrador):

```
npm install -g nodemon
```

Al instalarlo de forma global, se añadirá el comando `nodemon` en la misma carpeta donde residen los comandos `node` o `npm`. Para utilizarlo, basta con colocarnos en la carpeta del proyecto que queramos probar, y emplear este comando en lugar de `node` para lanzar la aplicación:

```
nodemon index.js
```

Automáticamente aparecerán varios mensajes de información en pantalla y el resultado de ejecutar nuestro programa. Ante cada cambio que hagamos, se reiniciará este proceso volviendo a ejecutarse el programa.

Para finalizar la ejecución de `nodemon` (y, por tanto, de la aplicación que estamos monitorizando), basta con pulsar Control+C en el terminal.

3.2. Desinstalar módulos globales

Del mismo modo, para desinstalar un módulo que se ha instalado de forma global, utilizaremos el comando:

```
npm uninstall -g nombre_modulo
```