

## Índice.

1. Los módulos en node.
2. Módulos de núcleo de node.
3. Utilizando nuestros propios módulos.
4. Módulos de terceros. NPM.
5. Instalar módulos locales a un proyecto.
6. Instalar módulos globales al sistema.

### 1. Los módulos en node.

Node.js es un framework muy modularizado, es decir, está subdividido en numerosos módulos, librerías o paquetes (a lo largo de estos apuntes utilizaremos estos tres términos indistintamente para referirnos al mismo concepto). De esta forma, sólo añadimos a nuestros proyectos aquellos módulos que necesitamos.

El propio núcleo de Node.js ya incorpora algunas librerías de uso habitual. Por ejemplo:

- http y https, para hacer que nuestra aplicación se comporte como un servidor web, o como un servidor web seguro o cifrado, respectivamente.
- fs para acceder al sistema de archivos
- utils, con algunas funciones de utilidad, tales como formato de cadenas de texto.

Para una lista detallada de módulos, podemos acceder en <https://nodejs.org/api/>. Es una API de todos los módulos incorporados en el núcleo de Node.js, con documentación sobre todos los métodos disponibles en cada uno.

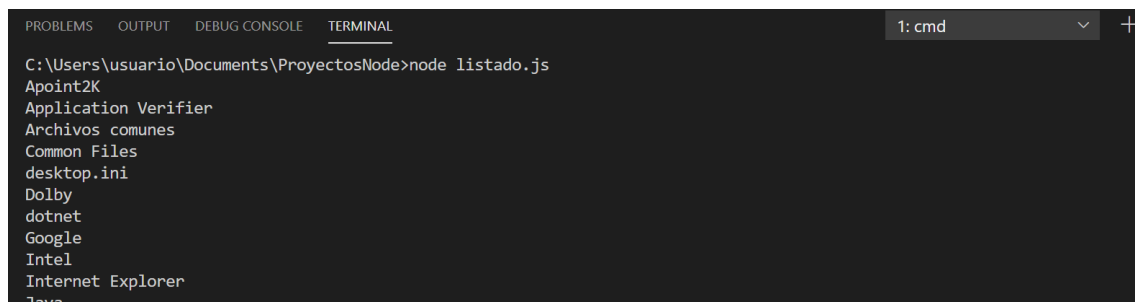
## 2. Módulos de núcleo de node.

Para utilizar cualquier módulo (propio de Node o hecho por terceras partes) en una aplicación es necesario incluirlo en nuestro código con la instrucción `require`. Recibe como parámetro el nombre del módulo a añadir, como una cadena de texto.

Por ejemplo, vamos a crear un archivo llamado `listado.js`. En él vamos a hacer un pequeño programa que utilice el módulo `fs` incorporado en el núcleo de Node para obtener un listado de todos los archivos y subcarpetas de una carpeta determinada. El código de este archivo puede ser más o menos así:

```
const ruta = 'C:/\Program Files';
const fs = require('fs');
fs.readdirSync(ruta).forEach(fichero => {console.log(fichero)});
```

Si ejecutamos este programa en el terminal (recordemos que podemos usar el terminal integrado de Visual Studio Code), obtendremos el listado de la carpeta indicada:

A screenshot of a terminal window within the Visual Studio Code interface. The terminal title bar shows '1: cmd'. The command prompt shows the path 'C:\Users\usuario\Documents\ProyectosNode' followed by the command 'node listado.js'. The output lists the contents of the 'C:\Program Files' directory, including 'Apoin2K', 'Application Verifier', 'Archivos comunes', 'Common Files', 'desktop.ini', 'Dolby', 'dotnet', 'Google', 'Intel', 'Internet Explorer', and 'Java'.

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL
C:\Users\usuario\Documents\ProyectosNode>node listado.js
Apoin2K
Application Verifier
Archivos comunes
Common Files
desktop.ini
Dolby
dotnet
Google
Intel
Internet Explorer
Java
```

## 3. Utilizando nuestros propios módulos.

Cuando estamos haciendo un proyecto mediano o grande es conveniente descomponer nuestra aplicación en diferentes módulos. Para hacer esto podemos crear un módulo que contendrá código independiente (instrucciones que no están dentro de una función) y funciones. El código independiente se ejecutará

directamente y las funciones tendrán que ser llamadas. Por ejemplo, dentro de una misma carpeta creamos dos ficheros: *utilidades.js* y *principal.js* dentro de la misma carpeta. El fichero *utilidades* contendrá:

```
//código independiente, si es necesario
console.log('Entrando en utilidades.js');

//funciones
let sumar = (num1, num2) => num1 + num2;
let restar = (num1, num2) => num1 - num2;

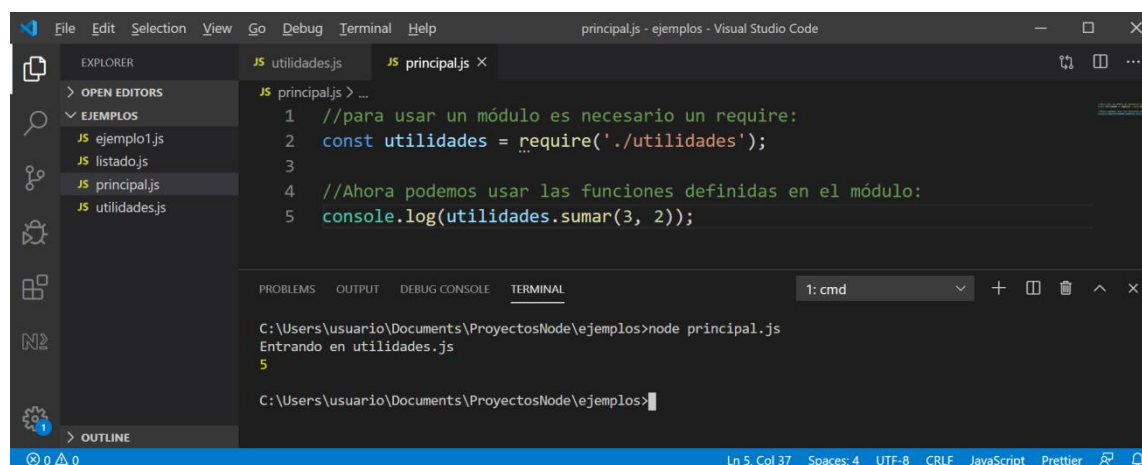
//para exportar funciones es necesario decirlo explícitamente
module.exports = {
  sumar: sumar,
  restar: restar
};
```

Y el fichero *principal*:

```
//para usar un módulo es necesario un require:
const utilidades = require('./utilidades');

//Ahora podemos usar las funciones definidas en el módulo:
console.log(utilidades.sumar(3, 2));
```

Si ejecutamos *principal*:



```
principal.js - ejemplos - Visual Studio Code

EXPLORER
  OPEN EDITORS
    EJEMPLOS
      ejemplo1.js
      listado.js
      principal.js
      utilidades.js

  principal.js > ...
    1 //para usar un módulo es necesario un require:
    2 const utilidades = require('./utilidades');
    3
    4 //Ahora podemos usar las funciones definidas en el módulo:
    5 console.log(utilidades.sumar(3, 2));

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL
  1: cmd
  C:\Users\usuario\Documents\ProyectosNode\ejemplos>node principal.js
  Entrando en utilidades.js
  5
  C:\Users\usuario\Documents\ProyectosNode\ejemplos>
```

El objeto `module.exports` admite tanto funciones como atributos o propiedades. Por ejemplo, podríamos definir una propiedad para almacenar el valor del número "pi":

```
module.exports = {  
  pi: 3.1416,  
  sumar: sumar,  
  restar: restar  
};
```

El ejemplo anterior funcionará siempre que ejecutemos la aplicación Node desde su misma carpeta:

```
node principal.js
```

Pero si estamos en otra carpeta y ejecutamos la aplicación desde allí...

```
C:\OtraCarpeta> node C:\ProyectosNode\Pruebas\PruebasRequire\principal.js
```

Entonces require hará referencia a la carpeta desde donde estamos ejecutando, y no encontrará el archivo "utilidades.js", en este caso. Para evitar este problema, podemos emplear la propiedad `__dirname`, que hace referencia a la carpeta del módulo que se está ejecutando ("principal.js", en este caso):

```
const utilidades = require(__dirname + '/utilidades');
```

#### **4. Módulos de terceros. 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 otros módulos o librerías que no tienen que ver con Node, como por ejemplo jQuery.

Si echamos un vistazo a la página principal de [nodejs.org](https://nodejs.org), hay una frase que dice que npm es el mayor ecosistema de librerías open-source del mundo, gracias a la comunidad de desarrolladores que hay detrás. Esto 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](https://npmjs.com).

## **5. Instalar módulos locales a un proyecto**

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:

```
npm init
```

Se 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, 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.

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
```

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 (<https://www.npmjs.com/package/lodash>). Para instalarlo, escribimos lo siguiente:

```
npm install lodash
```

Tras ejecutar el comando anterior, se habrá añadido el nuevo módulo en una subcarpeta llamada "node\_modules" dentro de nuestro proyecto.

Además se modifica el archivo "package.json" de configuración con el nuevo módulo incluido en el bloque de dependencias:

```
"dependencies": {  
  "lodash": "^4.17.4"  
}
```

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:

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

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

```
[ 2, 3 ]
```

Si decidimos subir nuestro proyecto a algún repositorio en Internet como Github o similares, o dejar que alguien se lo descargue para modificarlo después o simplemente para copiar el proyecto en otra carpeta, no es buena idea subir la carpeta "node\_modules", ya que puede llegar a ser muy grande. Por lo tanto, lo recomendable es no compartir la carpeta "node\_modules", y no es ningún problema hacer eso, ya que gracias al archivo "package.json" siempre podemos ejecutar el comando:

```
npm install
```

Lo que causará que se regenere la carpeta "node\_modules".

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

```
npm uninstall nombre_modulo
```

## **6. 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).