



# JavaScript sobre Node.js

**¿Sirve JavaScript para algo más que no sea hacer páginas web?**

JavaScript supuso una revolución cuando apareció a finales de los años noventa, introduciendo dinamismo en las webs e, incluso, la posibilidad de hacer operaciones con datos sin llamar a un servidor en Internet, lo que ahorraba tiempo y, por tanto, costes. Desde sus inicios, JavaScript ha sido protagonista de la programación web y lo sigue siendo. Y, por eso, quizás, le está costando mucho quitarse la etiqueta de lenguaje para la web, pero lo cierto es que hoy en día JavaScript sirve para mucho más.

Todo cambió en 2008, cuando Google lanzó el navegador de Internet Google Chrome, que incluía el motor de JavaScript V8 ❤️ ❤️ ❤️



¿Qué es un motor de JavaScript? Un lenguaje de programación necesita ser traducido para que la máquina que lo va a ejecutar lo pueda entender (¡las máquinas solo entienden ceros y unos!). El motor de JavaScript es el encargado de hacer la traducción de nuestro código fuente, nuestro programa, y de lanzar las instrucciones correspondientes para que la máquina las ejecute.

El caso es que Google liberó el proyecto del motor V8 como *open-source* y esto significó que desarrolladores de todo el mundo lo podían incluir en sus propios proyectos sin tener que pagar una licencia. Y aquí es donde precisamente entra Node.js.

**Node.js** es un entorno de ejecución de JavaScript cuyo corazón es el motor V8 de Google.



Un entorno de ejecución de JavaScript está constituido por una serie de programas que funcionan en colaboración con el motor de JavaScript para poder llevar a cabo las instrucciones que hemos definido en nuestro código.

Vamos a ver diferencia hay entre ejecutar un código JavaScript en un navegador como pueda ser Google Chrome y en el entorno de ejecución Node.js, teniendo en cuenta que ambos usan internamente el motor V8 de Google.

- Chrome, como cualquier otro navegador, está pensado para visitar páginas web en Internet y puede interactuar con el sistema operativo de nuestro ordenador hasta cierto punto. Por ejemplo, una limitación importante que un navegador nos impone es que no podremos acceder al sistema de ficheros del

ordenador: imagina que entrases en una web y se ejecutase un programa en JavaScript que borrase todos los ficheros de tu disco duro.  Sería peligroso dejar esa puerta abierta, por eso los navegadores están muy limitados en lo que al sistema operativo se refiere.

- Pero ¿sabías que un sistema gestor de bases de datos, como MongoDB o MySQL, no es más que un programa que organiza ficheros de texto? No tendría mayor problema en instalar uno de estos programas y darle acceso a mi sistema de ficheros, ¿verdad? Pues con Node.js podríamos crear un programa como estos, que acceda al sistema de ficheros de la máquina en el que esté instalado y a otras muchas funcionalidades que proporciona el sistema operativo.

O sea, el lenguaje de programación que usaríamos en ambos casos es JavaScript (una confusión muy extendida es pensar que Node.js es un lenguaje distinto a JavaScript: no es así). Lo que cambia, dependiendo del entorno, son el tipo de programas que puedo hacer:

- Sobre un navegador (por ejemplo, Google Chrome), instalado en un ordenador, podríamos tener un juego que funcionarse incluso sin conexión de red o wifi.
- Sobre un entorno de ejecución (por ejemplo, Node.js), instalado en un móvil, podría acceder a funciones del aparato como el acelerómetro o la cámara.
- Sobre un entorno de ejecución (por ejemplo, Node.js), instalado en un servidor, podríamos recibir peticiones a través de Internet y manejar el sistema de reservas de un hotel.

Como se ve en los ejemplos, dependiendo del entorno y dónde lo instalemos, podremos utilizar JavaScript para hacer unas cosas u otras. Pero lo que está más que claro a estas alturas es que puedo hacer mucho más que webs con este

lenguaje de programación, y esto en gran medida ha sido posible gracias al motor V8 de Google y a la aparición de Node.js.

## Ejecutando JavaScript en Node.js

Vamos a ver de forma práctica como las funcionalidades de JavaScript disponibles, van a depender del entorno de navegación que usemos.

Lo primero que vamos a hacer es ejecutar un código JavaScript muy sencillo en el navegador, el clásico "Hola mundo".

Para esto tenemos que crear en una carpeta un fichero de extensión html y otro de extensión js. Para enlazar el primero con el segundo vamos a usar la etiqueta de HTML <script>.

En el fichero js escribimos lo siguiente:

```
alert('Hola mundo');
```

Finalmente, abrimos el fichero HTML con el navegador y, si hemos hecho todo correctamente, veremos un mensaje emergente que nos dice: "Hola mundo".

Ahora vamos a probar a ejecutar el mismo código en Node.js (si no lo tienes instalado aún, [es el momento](#)).

Ejecutar nuestro código en Node.js es muy fácil, solo tenemos que abrir un terminal y escribir lo siguiente (suponiendo que nuestro código está en un fichero llamado index.js):

```
$ node index.js
```

Al pulsar intro, sin embargo, aparece un error en la terminal:

```
/Users/myuser/Documents/Code/index.js:1
alert('Hola mundo');
^
ReferenceError: alert is not defined
at Object.<anonymous> (/Users/myuser/Documents/Code/index.js:1:1)
at Module._compile (internal/modules/cjs/loader.js:1063:30)
at Object.Module._extensions..js (internal/modules/cjs/loader.js:1092:10)
at Module.load (internal/modules/cjs/loader.js:928:32)
at Function.Module._load (internal/modules/cjs/loader.js:769:14)
at Function.executeUserEntryPoint [as runMain] (internal/modules/run_main.js:72:12)
at internal/main/run_main_module.js:17:47
```

Cuando ejecutamos este mismo código en el navegador no dio ningún problema. ¿Qué está pasando?

La función `alert` provoca que aparezca una ventana emergente con el texto que hemos pasado por parámetro. Y esto está muy bien cuando estamos en un entorno visual, pero Node.js no tiene este tipo de interfaz. De hecho, si miramos la documentación de [la función `alert`](#), veremos que en realidad es un método del objeto `Window`, un objeto que solo tiene sentido en un navegador.

Finalmente, para comprobar que hay aún así funciones del lenguaje que son comunes tanto al navegador como a Node.js, vamos a cambiar el código que teníamos en nuestro fichero por el siguiente:

```
console.log('Hola mundo');
```

Ahora, tanto si ejecutamos el código en el navegador como si lo ejecutamos en Node.js, veremos que el resultado es el mismo: el texto "Hola mundo" aparece por la consola.



En una página web, para ver los mensajes de información y error de la consola de JavaScript, tenemos que abrir las herramientas del desarrollador en el navegador. Pero ¿dónde salen estos mensajes cuando ejecutamos sobre Node.js? Veremos que aparecen directamente en el terminal en el que estamos ejecutando nuestro fichero.

Una última cuestión sobre cómo se ejecuta un fichero JavaScript mediante Node.js: aunque para nosotros sea transparente, el entorno de ejecución envuelve el fichero que estamos ejecutando en una función (es lo que en otros lenguajes de programación llamamos *main function*). Y una de las ventajas de esto es que desde la línea de comandos podemos pasar parámetros como lo haríamos con una función cualquiera (¿recuerdas el objeto `arguments` de las funciones? es similar).

Si en mi fichero `main.js` tengo el siguiente código:

```
console.log(process.argv);
```

Al ejecutar el fichero desde la terminal, veré por la consola el contenido de un array con todos los elementos que he escrito en la línea de comandos:

```
$ node main.js one two three four five
[ 'node',
  '/home/avian/argvdemo/argv.js',
  'one',
  'two',
  'three',
```

```
'four',
'five' ]
```

Con este método podría pasarle a mi programa datos de inicio distintos cada vez que lo ejecute, y para usarlos en mi programa es tan fácil extraerlos del array.

## Módulos y sistemas de módulos

Antes de nada, ¿tienes clara la utilidad de una función?

Cuando nuestro código empieza a crecer vemos que hay partes de este que se repiten una y otra vez, así que creamos una función con las líneas del código que se repiten y así hacemos que nuestro programa sea más conciso, legible y mantenible.

Un ejemplo. Tenemos una página que se dibuja de forma dinámica con JavaScript, tomando datos de una API, y vemos que en varios sitios tenemos que añadir imágenes con pie de foto. Veamos el código de un par de estas imágenes:

```
const imageContainer1 = document.createElement('figure');
const image1 = document.createElement('img');
const imageCaption1 = document.createElement('figcaption');
image1.setAttribute('src', 'img/derivada.jpg');
image1.setAttribute('alt', 'Representación de una derivada');
imageCaption1.innerText = 'Representación de una derivada';
imageContainer1.appendChild(image1);
imageContainer1.appendChild(imageCaption1);

document.querySelector('body').appendChild(imageContainer1);

const imageContainer2 = document.createElement('figure');
const image2 = document.createElement('img');
const imageCaption2 = document.createElement('figcaption');
image2.setAttribute('src', 'img/integral.jpg');
image2.setAttribute('alt', 'Representación de una integral');
imageCaption2.innerText = 'Representación de una integral';
imageContainer2.appendChild(image2);
```

```
imageContainer2.appendChild(imageCaption2);
document.querySelector('body').appendChild(imageContainer2);
```

En el ejemplo anterior vemos que lo único que cambia del primer bloque al segundo son los datos, es el caso perfecto para utilizar una función:

```
function addPictureWithCaption(imageFile, imageCaptionText) {
    const imageContainer = document.createElement('figure');
    const image = document.createElement('img');
    const imageCaption = document.createElement('figcaption');
    image.setAttribute('src', imageFile);
    image.setAttribute('alt', imageCaptionText);
    imageCaption.innerText = imageCaptionText;
    imageContainer.appendChild(image);
    imageContainer.appendChild(imageCaption);
    document.querySelector('body').appendChild(imageContainer);
}
```

Con la función anterior, cada vez que necesitemos insertar una imagen con pie de foto será mucho más fácil:

```
//En una parte del código
addPictureWithCaption('img/derivada.jpg', 'Representación de una derivada');

//En otra parte
addPictureWithCaption('img/integral.jpg', 'Representación de una integral');
```

En definitiva, las funciones no son más que pequeños programas con una responsabilidad muy concreta, y su principal utilidad es que nos permiten reutilizar código.

Pues, bien, los módulos también son una forma de reutilizar código, pero con algunas diferencias:

- Un módulo no tiene por qué tener una responsabilidad tan concreta como una función, de hecho, un módulo suele estar compuesto (entre otros elementos) por varias funciones. Eso sí, las funciones de un módulo suelen girar en torno a unos mismos objetivos.
- Mientras la idea de una función es reutilizar código dentro de un mismo proyecto, la idea de un módulo va más allá: poder usar estas funcionalidades en aplicaciones distintas e, incluso, compartirlo con otros desarrolladores.
- Podemos usar funciones en nuestro código independientemente del entorno en el que vayamos a ejecutarlo, porque son un elemento básico del lenguaje, sin embargo, para usar módulos el entorno de ejecución debe implementar un sistema de módulos.

¿Sistema de módulos? ¡Otro concepto más! ¿Es que esto no se acaba nunca? La verdad es que no: cada concepto que aprendes te lleva a otros muchos nuevos, hay que ir acostumbrándose. 

Venga, vamos a ver qué hace un sistema de módulos.

Cuando nuestros programas solo corrían sobre el navegador, si queríamos dividir nuestro código en varios ficheros para organizarlo mejor, era tan fácil como luego incluir estos ficheros en el HTML en el orden apropiado:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
</head>
<body>
  <script src="API.js"></script>
  <script src="DOM.js"></script>
  <script src="main.js"></script>
```

```
</body>
</html>
```

Y si queríamos reutilizar el fichero `DOM.js` en otra web era tan fácil como copiarlo al nuevo proyecto e incluirlo en el HTML correspondiente. El problema es que esta forma de cargar los ficheros de JavaScript es exclusiva de los navegadores, así que, ¿qué pasa cuando quiero usar varios ficheros en Node.js? ¿cómo los puedo relacionar entre sí? Para eso sirve el sistema de módulos.

Node.js incluye por defecto un sistema de módulos llamado CommonJS. Vamos a ver cómo se definen módulos de este tipo.

Como hemos visto en uno de los primeros ejemplos, para ejecutar un fichero JavaScript con Node.js, tenemos que abrir una terminal e indicarle al entorno el nombre del fichero que queremos que execute. Esto significa que un programa que corra sobre Node.js siempre tendrá un fichero principal que llamamos punto de entrada:

```
$ node main.js
```

Para incluir un módulo en este fichero principal (que más arriba vemos que hemos llamado `main.js`), tenemos que referenciarlo mediante una sintaxis específica del sistema de módulos que estemos usando (en este caso, como nuestro entorno es Node.js, la sintaxis sería la de CommonJS):

```
//Esta es la referencia al módulo que está en otro fichero.
const addFunction = require('./math.js');
//El módulo está en este caso en el mismo directorio que el fichero principal.
//La función require devuelve en este caso una función
//que asignamos a una constante para poder utilizarla más adelante.

//La función del módulo se usa como cualquier otra:
console.log(addFunction(2, 5));
```

¿Y qué contenido tiene entonces el fichero `math.js`?

```
function add(value1, value2) {  
    return value1 + value2;  
}  
  
module.exports = add;
```

Un módulo es un universo autocontenido: ninguna variable, constante o función declarada dentro del fichero del módulo se podrá utilizar fuera de este salvo que se exporte. Dentro de un módulo también puedo requerir otros módulos, lo que permite gran flexibilidad.

Este es un ejemplo muy sencillito, lógicamente un módulo que nos proporcionase la capacidad de hacer operaciones matemáticas nos daría algo más que la posibilidad de sumar, pero es imprescindible entender lo más básico antes de profundizar.

## Módulos incluidos por defecto en Node.js

Los llamados *built-in modules* de Node.js nos proporcionan funcionalidad muy útil y para usarlos solo tenemos que requerirlos al principio de nuestro código. Igual que en el navegador si uso la función `alert` no necesito instalar nada, está disponible por defecto, Node.js tiene sus propias utilidades que están más relacionadas con el uso de funcionalidades a bajo nivel del sistema operativo, principalmente, con el sistema de ficheros, la comunicación a través de redes y el manejo de memoria.

Un ejemplo de cómo usar uno de estos módulos sería el siguiente:

```
//Aquí es donde incluimos el módulo,  
//que en este caso se llama FileSystem (fs)  
const fs = require('fs');  
  
//Con este módulo podemos leer un fichero de texto de nuestro ordenador.  
//Para hacerlo tenemos que usar un callback, pues la función que usaremos  
//es asíncrona:  
fs.readFile('./myTextFile.txt', function (err, data) {  
  if (err) {  
    console.log('Ha habido un error al leer el fichero: ', err);  
    return;  
  }  
  console.log(data);  
});
```

## Módulos de terceros

La verdadera potencia de los módulos es que se pueden compartir con la comunidad mundial de desarrolladores. Para utilizar estos módulos de terceros tenemos que descargarlos y vincularlos a nuestro proyecto (como veremos en el próximo epígrafe). Pero en todo caso es importante no perder de vista que al final la utilidad de los módulos, como ya hemos dicho, es no tener que reinventar la rueda: si una solución de software ya se ha probado, sabemos que funciona bien y no vemos la forma de mejorarla considerablemente, es mejor no perder el tiempo haciendo nuestra propia versión. Usa todo el código de terceros que puedas... siempre que sea de confianza.

## NPM: el gestor de paquetes por defecto de Node.js

Ahora que sabemos cómo se crea y utiliza un módulo (y también para qué sirve, claro), vamos a explicar cómo conseguir módulos de otros desarrolladores para no tener que estar una y otra vez reinventando la rueda: si una solución de software ya se ha hecho, sabemos que funciona bien y no vemos la forma de mejorarla considerablemente, lo ideal es no perder el tiempo haciendo nuestra propia

versión. La comunidad de desarrolladores de JavaScript es inmensa y muchos de ellos tienen la amabilidad de compartir sus soluciones en forma de módulos que se pueden incluir fácilmente en tu proyecto y de forma gratuita (*open-source*).

Usa todo el código de terceros que puedas... siempre que sea de confianza.

Por defecto, Node.js, utiliza el [repositorio NPM](#) e instala su cliente de línea de comandos. Si tienes instalado Node.js puedes comprobar qué versión tienes del entorno y del cliente NPM con los siguientes comandos en una terminal:

```
$ node -v  
v14.15.1  
$ npm -v  
6.14.8
```

Cuando vamos a utilizar módulos de terceros en nuestro proyecto, lo primero que debemos hacer es crear el fichero de configuración con el siguiente comando:

```
$ npm init -y
```

Al ejecutar este comando, vemos que se crea un fichero llamado `package.json` y que en el terminal nos aparece su contenido:

```
{  
  "name": "my-project",  
  "version": "1.0.0",  
  "description": "",  
  "main": "index.js",  
  "scripts": {  
    "test": "echo \\\"Error: no test specified\\\" && exit 1"  
  },  
  "keywords": []  
}
```

```
    "author": "",  
    "license": "ISC"  
}
```

Estos son los campos básicos, aunque podrían aparecer algunos más, por ejemplo, si nuestro tuviese un remoto de Git ya definido, o si tuviese un fichero README.md

Este fichero va a ser imprescindible si queremos usar librerías/módulos de terceros, pero, aunque no fuese el caso, nos sería de gran utilidad si queremos compartir nuestro proyecto con otros desarrolladores porque es un estándar para indicar detalles sobre un proyecto JavaScript.

Veamos esta estructura inicial:

- Los campos "name", "version", "description", "keywords", "author" y "license", son puramente informativos y opcionales, a no ser que vayamos a subir nuestro proyecto a los repositorios de NPM.
- El campo "main" indica el punto de entrada de nuestra aplicación. Recuerda que solo se puede ejecutar un fichero con Node.js, y desde ese fichero es desde donde cargaremos los demás módulos, sean propios, del entorno de ejecución o de terceros. En este ejemplo, el fichero index.js está en la misma carpeta/directorio que el fichero `package.json`, pero si no fuese así habría que indicar la ruta desde el fichero de configuración al punto de entrada de nuestra aplicación.
- El campo "scripts" nos permite definir alias para comandos que queramos ejecutar en la shell a menudo y que en muchos casos tendrán una serie de parámetros que sería trabajoso escribir cada vez. Por defecto, nos aparece un alias, "test", que debemos dejar porque indicará a los sistemas automáticos que no se han definido pruebas/test para este código.

Para continuar vamos a añadir un script más y a explicar cómo se ejecutan estos:

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

En general, los scripts se ejecutan de la siguiente forma en la terminal:

```
$ npm run nombreDelScript
```

Pero hay dos excepciones a este formato...

La ejecución del fichero principal (el punto de entrada), o sea, la puesta en marcha de nuestra aplicación, para la que nos podemos ahorrar la palabra "run":

```
$ npm start
```

Y para ejecutar las pruebas o tests del código (ya veremos más adelante qué es esto), para lo que tampoco necesitamos la palabra "run":

```
$ npm test  
Error: no test specified
```

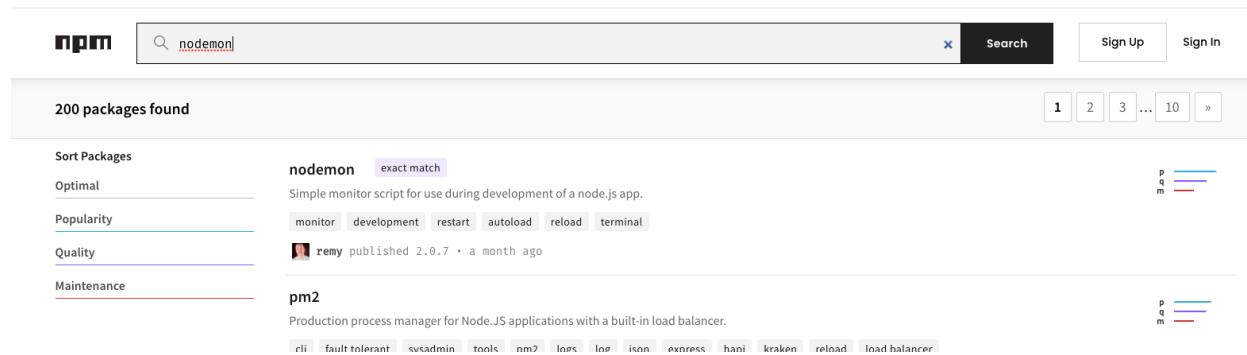
En todo caso, lo que nos tiene que quedar claro es que cuando usamos un script lo que estamos haciendo es llamar en línea de comandos al valor de este: por ejemplo, al poner `npm start`, lo que se ejecuta en línea de comandos en nuestra

caso es `node index.js`. Puede parecer que no ganamos gran cosa, pero `start` y `test` serán scripts que no solo usemos nosotros si no también, por ejemplo, el sistema de despliegue de ciertas plataformas (como GitLab o Heroku).

## Encontrando módulos de terceros

Cuando estemos en busca de módulos para resolver problemas de nuestra aplicación, podemos hacer una búsqueda directamente en Google, pero muy probablemente acabaremos en la página en la que están publicados los proyectos de la comunidad de NPM. Esta tiene su propio buscador, así que si sabes cómo buscar puedes hacerlo [directamente aquí](#).

Vamos a usar como ejemplo una librería llamada Nodemon, que nos vendrá muy bien como estemos en la fase de desarrollo creando nuestro propio servidor (el concepto es parecido al liveserver que nos permitía ver los cambios que estábamos haciendo en nuestro código sin necesidad de estar actualizando manualmente el navegador... pero en el servidor):



The screenshot shows the npm search interface. A search bar at the top contains the query 'nodemon'. Below it, a message says '200 packages found'. The results are listed under the heading 'Sort Packages'. The first result is 'nodemon' (exact match), which is described as a 'Simple monitor script for use during development of a node.js app.' It has tags: monitor, development, restart, autoload, reload, terminal. It was published by 'remy' (represented by a small profile icon) on 2.0.7 a month ago. The second result is 'pm2', described as a 'Production process manager for Node.JS applications with a built-in load balancer.' It has tags: cli, fault tolerant, svadmin, tools, pm2, lnes, log, ison, express, hapi, kraken, reload, load balancer.

**nodemon exact match**

Simple monitor script for use during development of a node.js app.

monitor development restart autoload reload terminal

 **remy** published 2.0.7 • a month ago

En este caso conocíamos el nombre de la librería y ha sido especialmente fácil. Pero también se puede ver en la captura que las librerías/modulos están clasificadas por varias etiquetas que nos hubieran servido también para encontrarla.

## Install

```
> npm i nodemon
```

♥ Fund this package

### ⚡ Weekly Downloads

**3.469.202**



### Version

**2.0.7**

### License

**MIT**

### Unpacked Size

**107 kB**

### Total Files

**43**

### Issues

**24**

### Pull Requests

**3**

### Homepage

**🔗 nodemon.io**

### Repository

**🔗 [github.com/remy/nodemon](https://github.com/remy/nodemon)**

### Last publish

**a month ago**

### Collaborators



Al entrar en el detalle de la librería, a la derecha vemos un resumen de estadísticas que nos pueden ayudar a detectar si esta es una librería de fiar. Desde luego, que haya varios millones de descargas a la semana ya te da una pista de que hay muchas personas que la están usando sin problemas. La licencia puede ser importante también a la hora de usar este módulo en un software comercial. Y, finalmente, es importante asegurarse de que alguien está manteniendo este software, que no se trata de un código obsoleto que hace años que nadie lo revisa.

Finalmente, al principio de todo, vemos el comando que habría que usar para instalar en tu proyecto este módulo:

```
$ npm i nodemon
```

En todo caso, esta sería la forma por defecto de instalar el módulo, pero a continuación veremos en mayor detalle cómo se instalan módulos en nuestro proyecto.

## Instalando módulos de terceros

Cuando instalamos librerías/módulos de terceros, estamos definiendo en nuestro proyecto lo que llamamos **dependencias**: o sea, a partir de ese momento, nuestro proyecto no funcionará sin este software de terceros.

Existen dos tipos de dependencias: las que formarán parte de nuestra aplicación final y las que solo vamos a utilizar durante la fase de desarrollo.

Nodemon, por ejemplo, es una dependencia de desarrollo, porque lo que hace es que no tengamos que volver a ejecutar nuestro proyecto cada vez que hacemos cambios en el código (lo que sería el comportamiento por defecto). Pero esto, cuando pasemos a producción (o sea, cuando pongamos el proyecto a disposición de los usuarios finales), no será necesario: el código no estará constantemente cambiando, como sucede en la fase de desarrollo.

Para instalar Nodemon, a pesar de lo que nos propone la página de NPM, lo ideal sería hacerlo de la siguiente forma:

```
$ npm install --save-dev nodemon
```

(el comando `install` tiene un diminutivo, como hemos visto en la página de NPM, que es simplemente su inicial, y el parámetro `--save-dev` tiene también un diminutivo que es `-D`, o sea, que podríamos también haber escrito `npm i -D nodemon`)

Así es como quedaría el `package.json` después de la instalación:

```
{
  "name": "my-project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "nodemon": "^2.0.7"
  }
}
```

Además, de este cambio en el `package.json`, vemos también que ha aparecido en nuestro proyecto un nuevo fichero, `package-lock.json`, que guarda un histórico de la evolución en las dependencias de tu proyecto, y también hay una nueva carpeta, `node_modules`, en la que están las librerías que hemos instalado con sus respectivas dependencias (porque los módulos también tienen sus dependencias 😊 ).

Si hubiésemos querido instalar un módulo como dependencia de producción, o sea, si estuviésemos instalando una librería que nuestra aplicación sí o sí va a necesitar, usaríamos el parámetro `--save` o `--save-prod`. Como ejemplo, vamos a instalar una librería de iconos:

```
$ npm install --save-prod @fortawesome/free-solid-svg-icons
```

(por defecto, si no usamos ningún parámetro, `-save`, `-save-prod` o `-save-dev`, se entiende que lo que estamos instalando es una dependencia de producción)

Así es como quedaría el `package.json` después de la instalación de una librería de producción:

```
{
  "name": "my-project",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "start": "node index.js",
    "test": "echo \\\"Error: no test specified\\\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "nodemon": "^2.0.7"
  },
  "dependencies": {
    "@fortawesome/fontawesome-free-solid-svg-icons": "^5.15.2"
  }
}
```

(se puede ver que las dependencias de desarrollo se registran en "devDependencies" y las de producción en "dependencies")



Técnicamente no sería imprescindible que todas las dependencias estuviesen en tu proyecto, existe la posibilidad de instalar modulos en tu equipo en vez de en el proyecto.

Para instalar un módulo de forma global lo haríamos usando el parámetro `-g`:

```
$ npm install -g nodemon
```

Como vamos a usar mucho Nodemon, podría parecer el perfecto candidato para instalarlo globalmente, sin embargo, no debemos instalar dependencias de nuestro proyecto de forma global. Solo se recomienda para los intérpretes de comandos (CLIs), como pueda ser el de Angular.

Aunque en el caso de los CLIs, también se podrían usar sin necesidad de instalarlos localmente, por medio del comando `npx`:

```
$ npx ng new my-first-project
```