

1. Introducción a Node.js

Parte I – Primeros pasos con Node.js

Despliegue de Aplicaciones Web

Nacho Iborra

IES San Vicente



Esta obra está licenciada bajo la Licencia Creative Commons Atribución-NoComercial-CompartirIgual 4.0 Internacional. Para ver una copia de esta licencia, visita <http://creativecommons.org/licenses/by-nc-sa/4.0/>

Índice de contenidos

1. Introducción a Node.js.....	1
1. ¿Qué es Node.js?.....	3
1.1. <i>Evolución de Javascript.....</i>	3
1.2. <i>Javascript en el servidor. El motor V8.....</i>	3
1.3. <i>Características principales de Node.js.....</i>	4
1.4. <i>¿Quién utiliza Node.js?.....</i>	5
2. Descarga e instalación.....	6
2.1. <i>Descargar e instalar Node.js.....</i>	6
2.2. <i>Primeras pruebas con el comando "node".....</i>	8
2.3. <i>Integrar Node.js en Visual Studio Code.....</i>	10
3. Las librerías o módulos.....	12
3.1. <i>Uso de "require" para incluir módulos.....</i>	13
3.2. <i>Uso de "require" para incluir nuestros propios archivos.....</i>	14
3.3. <i>Incluir módulos con "import".....</i>	18
4. El gestor de paquetes npm.....	19
4.1. <i>Instalar módulos locales a un proyecto.....</i>	20
4.2. <i>Instalar módulos globales al sistema.....</i>	24
4.3. <i>Orden de inclusión de los módulos.....</i>	25
5. Depuración de código.....	26
5.1. <i>Depurar desde Visual Studio Code.....</i>	26
5.2. <i>Depurar desde terminal.....</i>	27
5.3. <i>Depurar desde Google Chrome.....</i>	28
5.4. <i>Depurar con "nodemon".....</i>	29
6. Sobre require o module.exports.....	30
6.1. <i>exports y module.exports.....</i>	30

1. ¿Qué es Node.js?

Node.js es un entorno de ejecución en el lado del servidor construido utilizando el motor Javascript de Google Chrome, llamado V8. En este documento veremos cómo instalarlo y empezar a trabajar con él, pero antes conviene ser conscientes de lo que supone este paso en la historia del desarrollo web.

1.1. Evolución de Javascript

Como hemos comentado, Node.js es un entorno que emplea el lenguaje de programación Javascript y que se ejecuta en el lado del servidor. Esta afirmación puede resultar mundana, pero en realidad es algo sorprendente. Si echamos la vista atrás, el lenguaje Javascript ha pasado por 3 etapas o fases de expansión sucesivas:

1. La primera tuvo lugar con el primer apogeo de la web, allá por los años 90. Se comenzaban a desarrollar webs con HTML, y el Javascript que se empleaba entonces permitía añadir dinamismo a esas páginas, bien validando formularios, abriendo ventanas, o explorando el DOM (estructura de elementos de la página), añadiendo o quitando contenidos del mismo. Era lo que se conocía como *HTML dinámico* o DHTML.
2. La segunda etapa llegó con la incorporación de las comunicaciones asíncronas, es decir, con AJAX, más o menos a principios del siglo XXI. Se desarrollaron librerías como *Prototype* (primero) o *jQuery* (después) que abrieron todo un mundo nuevo de posibilidades con el lenguaje. Con ellas se podían actualizar fragmentos de la página, llamando desde Javascript a documentos del servidor, recogiendo la respuesta y pegándola en una zona concreta de la página, sin necesidad de recargarla por completo.
3. La tercera etapa ha llegado con la expansión de Javascript al lado del servidor. Hasta este momento sólo se utilizaba en los navegadores, por lo que sólo estábamos utilizando y viendo una versión reducida o restringida del lenguaje. Creíamos que Javascript sólo servía para validaciones, exploración del contenido HTML de una página o carga de contenidos en zonas concretas. Pero la realidad es que Javascript puede ser un lenguaje completo, y eso significa que podemos hacer con él cualquier cosa que se puede hacer con otros lenguajes completos, como Java o C#: acceder al sistema de ficheros, conectar con una base de datos, etc.

1.2. Javascript en el servidor. El motor V8

Bueno, vayamos asimilando esta nueva situación. Sí, Javascript ya no es sólo un lenguaje de desarrollo en el cliente, sino que se puede emplear también en el servidor. Pero... ¿cómo? En el caso de Node.js, como hemos comentado, lo que se hace es utilizar de manera externa el mismo motor de ejecución que emplea Google Chrome para compilar y ejecutar Javascript en el navegador: el motor V8. Dicho motor se encarga de compilar y ejecutar el código Javascript, transformándolo en código más rápido (código máquina). También se encarga de colocar los elementos necesarios en memoria, eliminar de ella los elementos no utilizados (*garbage collection*), etc.

V8 está escrito en C++, es open-source y de alto rendimiento. Se emplea en el navegador Google Chrome y "variantes" como Chromium (adaptación de Chrome a sistemas Linux), además de en Node.js y otras aplicaciones. Podemos ejecutarlo en sistemas Windows (XP o posteriores), Mac OS X (10.5 o posteriores) y Linux (con procesadores IA-32, x64, ARM o MIPS).

Además, al estar escrito en C++ y ser de código abierto, podemos extender las opciones del propio Javascript. Como hemos comentado, inicialmente Javascript era un lenguaje concebido para su ejecución en un navegador. No podíamos leer un fichero de texto local, por ejemplo. Sin embargo, con Node.js se ha añadido una capa de funcionalidad extra a la base proporcionada por V8, de modo que ya es posible realizar estas tareas, gracias a que con C++ sí podemos acceder a los ficheros locales, o conectar a una base de datos.

Por supuesto, como cualquier otro motor Javascript, V8 se ajusta al estándar ECMAScript sobre cómo debe comportarse el código generado (es decir, qué se espera que haga cada función registrada en el estándar cuando se le llame).

1.2.1. Requisitos para que Javascript pueda correr en el servidor

¿Qué características tienen lenguajes como PHP o JSP que no tenía Javascript hasta la aparición de Node.js, y que les permitían ser lenguajes en entorno servidor? Quizá las principales sean:

- Disponen de mecanismos para acceder al sistema de ficheros, lo que es particularmente útil para leer ficheros de texto, o subir imágenes al servidor, por poner dos ejemplos.
- Disponen de mecanismos para conectar con bases de datos
- Permiten comunicarnos a través de Internet (el estándar ECMAScript no dispone de estos elementos para Javascript)
- Permiten aceptar peticiones de clientes y enviarles respuestas a dichas peticiones

Nada de esto era posible en Javascript hasta la aparición de Node.js. Este nuevo paso en el lenguaje le ha permitido, por tanto, conquistar también el otro lado de la comunicación cliente-servidor para las aplicaciones web.

1.2.2. Consecuencia: un único lenguaje de desarrollo web

La consecuencia lógica de utilizar Node.js en el desarrollo de servidor es que, teniendo en cuenta que Javascript es también un lenguaje de desarrollo en el cliente, nos hará falta conocer un único lenguaje para el desarrollo completo de una aplicación web. Antes de que esto fuera posible, era indispensable conocer, al menos, dos lenguajes (Javascript para la parte de cliente y PHP, JSP, ASP.NET u otro lenguaje para la parte del servidor).

1.3. Características principales de Node.js

Entre las principales características que ofrece Node.js, podemos destacar las siguientes:

- Node.js ofrece una API **asíncrona** (es decir, que no bloquea el programa principal cuando llamamos a sus métodos esperando una respuesta) y **dirigida por eventos** (lo que permite recoger una respuesta cuando ésta se produce, sin dejar al programa esperando por ella). Comprenderemos mejor estos conceptos más adelante, cuando los pongamos en práctica.

- La ejecución de código es **muy rápida** (recordemos que se apoya en el motor V8 de Google Chrome)
- Modelo **monohilo** pero muy **escalable**. Se tiene un único hilo atendiendo peticiones de clientes, a diferencia de otros servidores que permiten lanzar hasta N hilos en paralelo atendiendo peticiones. Sin embargo, la API dirigida por eventos permite atender múltiples peticiones por ese único hilo, consumiendo muchos menos recursos que los sistemas multihilo.
- Se elimina la necesidad de **cross-browser**, es decir, de desarrollar código Javascript que sea compatible con todos los navegadores, que es a lo que el desarrollo en el cliente nos tiene acostumbrados. En este caso, sólo debemos preocuparnos de que nuestro código Javascript sea correcto para ejecutarse en el servidor.

1.4. ¿Quién utiliza Node.js?

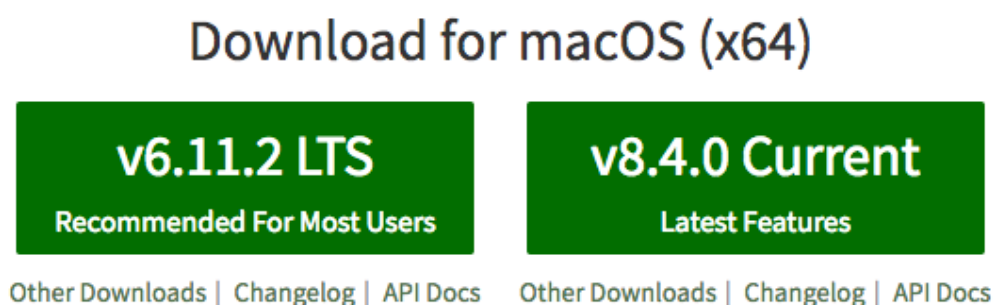
Es cierto que la mayoría de tecnologías emergentes suelen tardar un tiempo hasta tener buena acogida en nuestro país, salvo algunas pocas empresas pioneras. Pero sí hay varias empresas extranjeras, algunas de ellas de un peso relevante a nivel internacional, que utilizan *Node.js* en su desarrollo. Por poner algunos ejemplos representativos, podemos citar a Netflix, PayPal, Microsoft o Uber, entre otras.

2. Descarga e instalación

Antes de comenzar a dar nuestros primeros pasos con Node.js, vamos a instalarlo dependiendo de nuestro sistema operativo, comprobar que la instalación y versión son correctas, y editar un primer programa básico para probar con nuestro IDE.

2.1. Descargar e instalar Node.js

Para descargar Node.js, en general debemos acudir a su web oficial (nodejs.org), y hacer clic sobre el enlace de descarga que aparecerá, que ya está preparado para el sistema operativo que estamos utilizando. Actualmente, en la página principal se nos ofrecen dos versiones alternativas:



La versión LTS es la recomendada para la mayoría de usuarios. Es una versión algo obsoleta (ya que no es la última), pero ofrece soporte a largo plazo (LTS, *Long Term Support*) y casi todas las funcionalidades añadidas a Node.js, salvo las que aparecen en la última versión disponible, que es la otra opción que se nos ofrece para los que quieran probar las últimas novedades.

En nuestro caso, y dado que estamos en un módulo donde vamos a aprender cuantas más cosas mejor sobre Node.js, instalaremos la última versión (8.4.0 a la hora de hacer estos apuntes, aunque hablaremos de la versión 8 en general). Es posible que en la vida real nos encontremos instalada la última versión LTS en las empresas en las que estemos, o nos interese instalar esta otra versión para garantizarnos un soporte a largo plazo.

Así pues, hacemos clic en en el enlace a la última versión (enlace derecho) y descargamos el paquete:

- Para sistemas **Windows** el paquete es un instalador (archivo *.msi*) que podemos directamente ejecutar para que se instale. Aceptamos el acuerdo de licencia, y confirmamos cada paso del asistente con los valores por defecto que aparezcan.



- Para sistemas **Mac OS X**, el paquete es un archivo **.pkg** que podemos ejecutar haciendo doble click en él, y seguir los pasos del asistente como en Windows.



- Para sistemas **Linux**, el paquete que se descarga es un archivo con extensión **.tar.xz**. Sin embargo, si consultamos información por Internet, se recomienda instalar Node.js desde un repositorio. Para ello, y contando con instalar la versión 8 que es la última actualmente, podemos escribir estos comandos en un terminal en modo **root**, es decir, anteponiendo el comando **sudo** a cada comando que escribamos, o iniciando todo este proceso con el comando **su**, si conocemos la contraseña de **root**:

```
apt-get update
```

```
apt-get upgrade
```

```
apt-get install curl
```

```
curl -sL https://deb.nodesource.com/setup_8.x | bash -
```

```
apt-get install -y nodejs
```

NOTA: en el caso de tener el comando **curl** ya previamente instalado, sólo será necesario ejecutar las dos últimas instrucciones.

En cualquier caso, se instalará tanto el comando `node` que permitirá interactuar y probar nuestras aplicaciones Node.js, como el gestor de paquetes `npm` (*Node Package Manager*), cuya utilidad comentaremos más adelante.

2.1.1. Actualizar desde una versión previa

La actualización desde versiones previas es tan sencilla como descargar el paquete de la nueva versión y ejecutarlo. Automáticamente se sobrescribirá la versión antigua con la nueva. En el caso de Linux, podemos repetir la secuencia de comandos anterior cambiando el paquete de la nueva versión (`setup_8.x`) por el de la versión que sea. También podemos instalar un gestor de versiones de Node llamado `nvm`, que nos ayudará a descargar la versión que queramos, e incluso a simultanear varias versiones y elegir en cada momento cuál usar. Esta opción no la trataremos en este módulo.

2.2. Primeras pruebas con el comando "node"

Una vez instalado Node.js, podemos comprobar la instalación y la versión instalada desde línea de comandos, con el comando `node`. Para ello, abrimos un terminal en el sistema en que estemos y escribimos este comando:

```
node -v
```

También podemos utilizar `node --version` en su lugar. En ambos casos, nos deberá aparecer la versión que hemos instalado, que en nuestro caso será algo parecido a esto:

```
v8.4.0
```

IMPORTANTE: debemos asegurarnos de que podemos escribir este comando y obtener la salida esperada antes de continuar, ya que de lo contrario tampoco podremos ejecutar nuestros programas Node.

Este comando `node` también se emplea para ejecutar desde el terminal un archivo fuente Node.js. Por ejemplo, podemos editar un programa llamado *prueba.js* con este contenido:

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

Y después ejecutarlo con este comando (desde la misma carpeta donde tengamos el archivo fuente):

```
node prueba.js
```

La salida en este caso será:

```
Hola mundo
```

2.2.1. Uso de "node" como intérprete de comandos

Si escribimos el comando `node` a secas en el terminal, aparecerá un "prompt" con una flecha a la derecha, que indica que hemos entrado en el intérprete de comandos de Node.

```
node
```

```
>
```

Si escribimos cualquier instrucción Javascript de las que veremos en el curso en este terminal, se evaluará y ejecutará directamente:

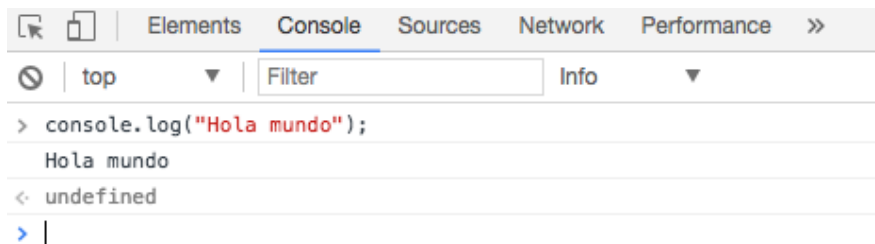
```
> console.log("Hola mundo");
```


Hola mundo

Es una forma simple de probar ciertas instrucciones sencillas. Para cerrar este intérprete, podemos pulsar dos veces la combinación de teclas Control+C, o escribir el comando `process.exit(0)`.

2.2.2. Similitudes entre este terminal y la consola de Chrome

Si alguna vez habéis depurado el funcionamiento o el diseño de una aplicación web en Chrome, habréis ido al menú *Más herramientas > Herramientas para desarrolladores*. Abre un panel donde podemos examinar el código HTML y CSS de nuestra web, y hay una pestaña llamada *Console* para interactuar con el motor V8. Ahí aparecen errores de ejecución Javascript, pero también podemos escribir comandos, de la misma forma que en el terminal node que hemos abierto antes:



Sin embargo, hay algunas diferencias en el comportamiento de ambas consolas. La que ofrece Google Chrome tiene los objetos `window` y `document`, que os sonarán si habéis escrito código Javascript para ser ejecutado en el navegador. Si escribimos cualquiera de estos dos elementos en el terminal de Chrome y pulsamos Intro, aparecerán todas las propiedades que tienen definidas, y sus valores.

```
> document
< #document
  <!DOCTYPE html>
  <html lang="es-es" class>
    <head>...</head>
    <body id="es" class="course-taking udemy no-keyboard-navigation-in-use
pageloaded no-header-or-footer ud-angular-loaded" data-module-id="course-
taking-v4" data-module-name="course-taking-v4/app">...</body>
  </html>
> |
```

Por su parte, el terminal node no dispone de estos dos objetos, ya que no estamos en una ventana de navegador, ni generando código HTML. En su lugar, tenemos disponibles los elementos `global` (equivalente a `window`, pero para terminal), y `process` (que hace referencia al proceso Node actualmente en ejecución y los elementos que contiene). Si escribimos cualquiera de estos dos elementos en nuestro terminal node, podemos examinar su contenido.

```
> process
process {
  title: 'node',
  version: 'v8.2.1',
  moduleLoadList:
    [ 'Binding contextify',
      'Binding natives',
      'Binding config',
      'NativeModule events',
      'Binding async_wrap',
      'Binding icu',
      'NativeModule util',
```

2.3. Integrar Node.js en Visual Studio Code

Visual Studio Code ofrece una integración muy interesante con Node.js, de manera que podemos editar, ejecutar y depurar nuestras aplicaciones desde este IDE. Veamos qué pasos seguir para ello.

2.3.1. Preparar el espacio de trabajo

Cada proyecto Node que hagamos irá contenido en su propia carpeta y, por otra parte, Visual Studio Code y otros editores similares que podamos utilizar (como Atom o Sublime) trabajan por carpetas (es decir, les indicamos qué carpeta abrir y nos permiten gestionar todos los archivos de esa carpeta). Por lo tanto, lo primero que haremos será crear una carpeta llamada "ProyectosNode" en nuestro espacio de trabajo (por ejemplo, en nuestra carpeta personal). Dentro de esta carpeta, crearemos tres subcarpetas:

- **Pruebas**, donde guardaremos todos los proyectos de prueba y ejemplo que hagamos
- **Ejercicios**, donde almacenaremos los ejercicios propuestos de cada sesión
- **Practicas**, donde irán las prácticas finales de los temas que correspondan

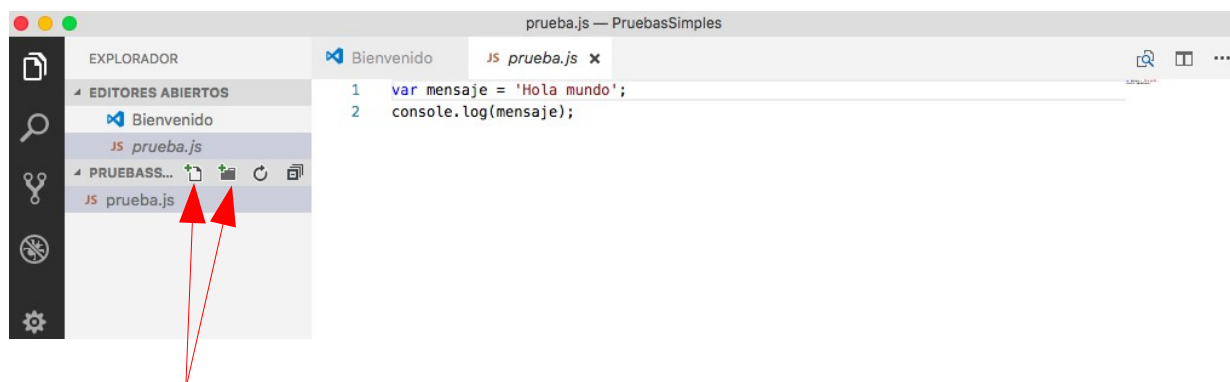
La estructura de carpetas quedará entonces como sigue:

- ProyectosNode
 - Pruebas
 - Ejercicios
 - Practicas

2.3.2. Crear y editar un proyecto básico

Dentro de la carpeta *ProyectosNode/Pruebas*, vamos a crear otra subcarpeta llamada "PruebasSimples" donde definiremos pequeños archivos para probar algunos conceptos básicos, especialmente en las primeras sesiones.

Una vez creada la carpeta, abrimos Visual Studio Code y vamos al menú *Archivo > Abrir carpeta* (o *Archivo > Abrir...*, dependiendo de la versión de Visual Studio Code que tengamos). Elegimos la carpeta "PruebasSimples" dentro de *ProyectosNode/Pruebas* y se abrirá en el editor. De momento está vacía, pero desde el panel izquierdo podemos crear nuevos archivos y carpetas. Para empezar, vamos a crear un archivo "prueba.js" como el de un ejemplo previo, con el código que se muestra a continuación:

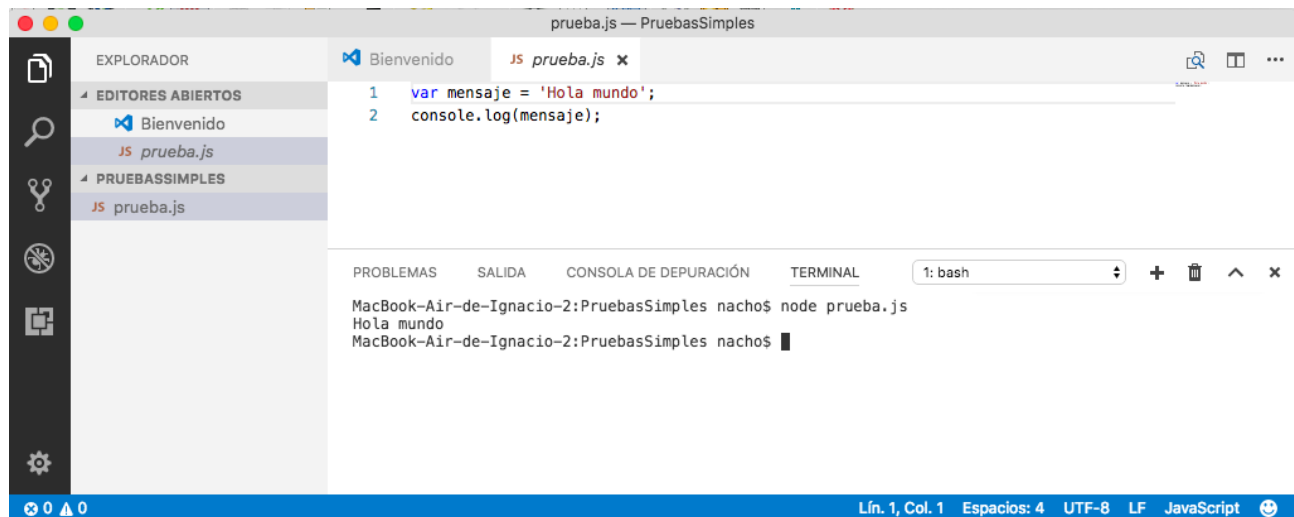


Crear nuevos archivos y carpetas en la actual

2.3.3. Ejecutar un archivo Node desde Visual Studio Code

Si quisiéramos ejecutar el programa anterior con lo visto hasta ahora, deberíamos abrir un terminal, navegar hasta la carpeta del proyecto y ejecutar el comando `node prueba.js`, como hicimos en un ejemplo anterior.

Sin embargo, Visual Studio Code cuenta con un terminal incorporado, que podemos activar yendo al menú *Ver > Terminal integrada*. Aparecerá en un panel en la zona inferior. Observemos cómo automáticamente dicho terminal se sitúa en la carpeta de nuestro proyecto actual, por lo que podemos directamente escribir `node prueba.js` en él y se ejecutará el archivo, mostrando el resultado en dicho terminal:



The screenshot shows the Visual Studio Code interface. On the left, the Explorer sidebar shows the project structure with 'prueba.js' selected. The main editor area displays the code in 'prueba.js':

```
1 var mensaje = 'Hola mundo';  
2 console.log(mensaje);
```

At the bottom, the integrated terminal is open, showing the command `node prueba.js` being executed in a bash shell. The output of the command is `Hola mundo`.

```
MacBook-Air-de-Ignacio-2:PruebasSimples nacho$ node prueba.js  
Hola mundo  
MacBook-Air-de-Ignacio-2:PruebasSimples nacho$
```

The status bar at the bottom indicates the current file is 'prueba.js' at line 1, column 1, using UTF-8 encoding and LF line endings.

3. Las librerías o módulos

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 necesitemos.

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.
- ... etc. Para una lista detallada de módulos, podemos acceder [aquí](#). Es una API de todos los módulos incorporados en el núcleo de Node.js, con documentación sobre todos los métodos disponibles en cada uno. Por ejemplo, aquí podemos ver la documentación sobre el método `readdirSync` del módulo `fs`, que utilizaremos en un ejemplo a continuación:

fs.readdirSync(path[, options])

#

► History

- `path` `<string>` | `<Buffer>` | `<URL>`
- `options` `<string>` | `<Object>`
 - `encoding` `<string>` Default: `'utf8'`

Synchronous `readdir(3)`. Returns an array of filenames excluding `'.'` and `'..'`.

The optional `options` argument can be a string specifying an encoding, or an object with an `encoding` property specifying the character encoding to use for the filenames passed to the callback. If the `encoding` is set to `'buffer'`, the filenames returned will be passed as `Buffer` objects.

Además, existen numerosos módulos hechos por terceros que pueden ser añadidos y utilizados en nuestras aplicaciones, como por ejemplo el módulo *mongoose* para acceso a bases de datos MongoDB, o el módulo *express* para incorporar el framework Express.js a nuestro proyecto, como veremos en temas posteriores. Estos módulos de terceros se instalan a través del gestor npm que explicaremos en breve.

Javascript no siempre ha sido modular. Node.js y ES6 lo han hecho posible

Hasta la aparición de Node.js y las últimas especificaciones de ECMAScript (ES6), Javascript no era un lenguaje modularizado, es decir, no era posible escribir fragmentos de código reutilizables e independientes del resto de nuestro código Javascript. Sí podíamos dividir el código en diferentes fuentes, pero podíamos tener problemas si, por ejemplo, había dos funciones en dos archivos fuente diferentes que se llamaran igual.

Con ECMAScript2015 (o ES6) se ha añadido de forma oficial esta modularización a través de un estándar llamado *CommonJS*, y en entornos como Node.js se utiliza de forma habitual. Sin embargo, la forma de incluir módulos en ES6 y en Node es diferente, y utilizaremos esta última durante el módulo. Pero con los años será cada vez más común que ambas formas coexistan, y que el estándar ES6 se vaya generalizando.

3.1. Uso de "require" para incluir módulos

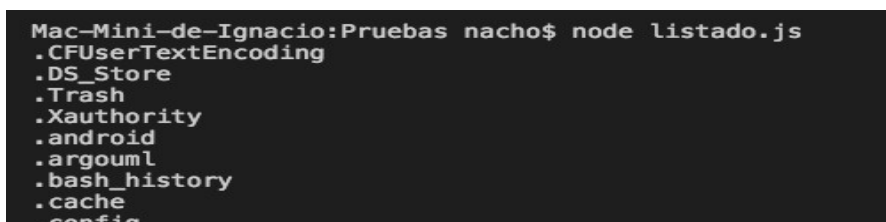
Para utilizar cualquier módulo (propio de Node o hecho por terceras partes) en una aplicación es necesario incluirlo en nuestro código con la instrucción **require**. Recibe como parámetro el nombre del módulo a añadir, como una cadena de texto. Veamos un par de ejemplos de uso.

3.1.1. Ejemplo sencillo: uso del módulo "fs" para listar ficheros

Por ejemplo, vamos a crear un archivo llamado "listado.js" en nuestro proyecto de "PruebasSimples". 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 = '/Users/nacho';
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. Antes de ejecutarlo, recuerda cambiar el valor de la constante `ruta` en el código por el de una carpeta válida en tu sistema:



```
Mac-Mini-de-Ignacio:Pruebas nacho$ node listado.js
.CFUserTextEncoding
.DS_Store
.Trash
.Xauthority
.android
.argouml
.bash_history
.cache
.config
```

Notar que en el código hemos declarado dos constantes (`const`), en lugar de variables (`var`), ya que no necesitamos manipular o modificar el código que se almacenará en ellas una vez cargadas (no vamos a cambiar la ruta, ni el contenido del módulo `fs` en nuestro código). En ejemplos que podáis encontrar en Internet, es habitual, no obstante, el uso de `var` en estos casos.

Ejercicio 1

Crea una carpeta llamada "Tema1_SaludoUsuario" en tu espacio de trabajo, dentro de la carpeta de "Ejercicios". Añade un archivo llamado "saludo.js". Echa un vistazo en la [API](#) de Node al módulo "os", y en concreto al método `userInfo`. Utilízalo para hacer un programa que salude al usuario que ha accedido al sistema operativo. Por ejemplo, si el usuario es "nacho", debería decir "Hola nacho". Ejecuta el programa en el terminal para comprobar su correcto funcionamiento.

AYUDA: el método `userInfo` devuelve un *objeto* con varias propiedades del usuario que ha accedido. Para obtener el nombre del usuario, deberemos acceder a la propiedad `username`.

AYUDA: el método `console.log` admite un número indefinido de parámetros, y los concatena uno tras otro con un espacio. De forma que estas dos líneas son equivalentes:

```
console.log("Hola " + nombre);
console.log("Hola", nombre);
```

3.2. Uso de "require" para incluir nuestros propios archivos

También podemos utilizar `require` para incluir un archivo nuestro dentro de otro, de forma que podemos (debemos) descomponer nuestra aplicación en diferentes archivos, lo que la hará más fácil de mantener.

Por ejemplo, vamos a crear un proyecto llamado "PruebasRequire" en nuestra carpeta de "Pruebas", con dos archivos de momento: uno llamado "principal.js" que tendrá el código principal de funcionamiento del programa, y otro llamado "utilidades.js" con una serie de funciones o propiedades auxiliares. Desde el archivo "principal.js", podemos incluir el de utilidades con la instrucción `require`, del mismo modo que lo hicimos antes, pero indicando la ruta relativa del archivo a incluir. En este caso quedaría así:

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

Es posible también suprimir la extensión del archivo en el caso de archivos Javascript, por lo que la instrucción anterior sería equivalente a esta otra (Node sobreentiende que se trata de un archivo Javascript):

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

El contenido del archivo "utilidades.js" debe tener una estructura determinada. Si, por ejemplo, el archivo tiene este contenido:

```
console.log('Entrando en utilidades.js');
```

Entonces el mero hecho de incluirlo con `require` mostrará por pantalla el mensaje "Entrando en utilidades.js" al ejecutar la aplicación. Es decir, cualquier instrucción directa que contenga el archivo incluido se ejecuta al incluirlo. Lo normal, no obstante, es que este archivo no contenga instrucciones directas, sino una serie de propiedades y métodos que puedan ser accesibles desde el archivo que lo incluye.

Vamos a ello, supongamos que el archivo "utilidades.js" tiene unas funciones matemáticas para sumar y restar dos números y devolver el resultado. Algo así:

```
function sumar(num1, num2) {  
    return num1 + num2;  
}  
  
function restar(num1, num2) {  
    return num1 - num2  
}
```

Lo lógico sería pensar que, al incluir este archivo con `require` desde "principal.js", podamos acceder a las funciones `sumar` y `restar` que hemos definido... pero no es así.

3.2.1. Exportando contenido con `module.exports`

Para poder hacer los métodos o propiedades de un archivo visibles desde otro que lo incluya, debemos añadirlos como elementos del objeto **`module.exports`**. Así, las dos funciones anteriores se deberían definir mediante funciones anónimas asociadas a un nombre, de esta forma:

```
module.exports.sumar = function(num1, num2) {  
    return num1 + num2;  
};
```

```
module.exports.restar = function(num1, num2) {  
    return num1 - num2  
};
```

Es habitual definir un objeto en `module.exports`, y añadir dentro todo lo que queramos exportar. De esta forma, tendremos por un lado el código de nuestro módulo, y por otro la parte que queremos exportar. El módulo quedaría así, en este caso:

```
function sumar(num1, num2) {  
    return num1 + num2;  
}  
  
function restar (num1, num2) {  
    return num1 - num2  
}
```

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

En cualquier caso, ahora sí podemos utilizar estas funciones desde el programa principal:

```
const utilidades = require('./utilidades');  
console.log(utilidades.sumar(3, 2));
```

Notar que 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  
};
```

... y acceder a ella desde el programa principal:

```
console.log(utilidades.pi);
```

3.2.2. Un adelanto: las "arrow functions"

Vamos a introducir un concepto que se ha vuelto muy habitual con ES6, y que utilizaremos a menudo en estos apuntes. Se trata de una notación alternativa para definir métodos, las llamadas *arrow functions* (funciones flecha), que emplean una expresión lambda para definir los parámetros por un lado (entre paréntesis) y el código de la función por otro entre llaves, separados por una flecha. Se prescinde de la palabra reservada *function* para definir las. Las dos funciones anteriores se definirían así:

```
var sumar = (num1, num2) => {  
    return num1 + num2;  
};  
  
var restar = (num1, num2) => {  
    return num1 - num2  
}
```


El programa principal no se vería alterado (llamaríamos a las funciones de la misma forma que con la definición tradicional). De hecho, el código anterior puede simplificarse aún más: en el caso de que la función simplemente devuelva un valor, se puede prescindir de las llaves y de la palabra `return`, quedando así:

```
var sumar = (num1, num2) => num1 + num2;  
var restar = (num1, num2) => num1 - num2;
```

La diferencia entre las *arrow functions* y la nomenclatura tradicional es que con las *arrow functions* no podemos acceder al elemento `this`, o al elemento `arguments`, que sí están disponibles con las funciones anónimas o tradicionales. Así que, en caso de necesitar hacerlo, deberemos optar por una función anónima en este caso.

Ejercicio 2

Crea una carpeta llamada "Tema1_VerificarUsuario" en tu espacio de trabajo, en la carpeta de "Ejercicios". Añade un archivo llamado "utilidades_os.js" y otro llamado "principal.js". Dentro de "utilidades_os.js" haz lo siguiente:

- Carga el módulo "os" (con `require`)
- Exporta una propiedad llamada "loginUsuario" que almacene el login del usuario que accedió al sistema, de forma similar a como lo obtuviste en un ejercicio anterior.
- Exporta un método llamado "esUsuario" que reciba como parámetro una cadena y devuelva un booleano dependiendo de si la cadena coincide con la propiedad "loginUsuario" o no.

En el archivo "principal.js", haz lo siguiente:

- Incluye el archivo "utilidades_os.js" hecho anteriormente (con `require`)
- Llama al método "esUsuario" pasándole como parámetro el nombre "pepe", y escribe por pantalla un texto indicando si es ése el usuario logueado o no (en función de lo que te devuelva la llamada al método).
- Utiliza la propiedad "loginUsuario" para mostrar por pantalla quién es el auténtico usuario logueado.

Al ejecutar el programa principal, deberá mostrarte estos mensajes (suponiendo que tu usuario sea "nacho"):

```
El usuario no es 'pepe'
```

```
El usuario correcto es 'nacho'
```

3.2.3. Incluir carpetas enteras

En el caso de que nuestro proyecto contenga varios módulos, es recomendable agruparlos en carpetas, y en este caso es posible incluir una carpeta entera de módulos, siguiendo una nomenclatura específica. Los pasos a seguir son:

- Añadir todos los módulos (ficheros `.js`) que queramos dentro de la carpeta deseada
- Crear en esa carpeta un archivo llamado "index.js". Este será el archivo que se incluirá en nombre de toda la carpeta
- Dentro de este archivo "index.js", incluir (con `require`) todos los demás módulos de la carpeta.

- Desde el programa principal (u otro lugar que necesite incluir la carpeta entera), incluir el nombre de la carpeta. Automáticamente se localizará e incluirá el archivo "index.js", con todos los módulos que éste haya incluido dentro.

Veamos un ejemplo: vamos a nuestra carpeta "PruebasRequire" y crea una carpeta llamada "idiomas". Dentro, crea estos tres archivos, con el siguiente contenido:

- Archivo "es.js":

```
module.exports = {
  saludo : "Hola"
};
```
- Archivo "en.js":

```
module.exports = {
  saludo : "Hello"
};
```
- Archivo "index.js":

```
const en = require('./en');
const es = require('./es');

module.exports = {
  es : es,
  en : en
};
```

Ahora, en la carpeta raíz de "PruebasRequire" crea un archivo llamado "saludo_idioma.js", con este contenido:

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

console.log("English:", idiomas.en.saludo);
console.log("Español:", idiomas.es.saludo);
```

Como puedes ver, desde el archivo principal sólo hemos incluido la carpeta, y con eso automáticamente incluimos el archivo "index.js" que contiene, que a su vez incluye a todos los demás. Una vez hecho esto, y tal y como hemos exportado las propiedades en "index.js", podemos acceder al saludo en cada idioma.

3.2.4. Incluir archivos JSON

Los archivos JSON son especialmente útiles, como veremos, para definir cierta configuración básica (no encriptada) en las aplicaciones, además de para enviar información entre partes de la aplicación (lo que veremos también más adelante). Por ejemplo, y siguiendo con el proyecto anterior, podríamos sacar a un archivo JSON el texto del saludo en cada idioma. Añadamos un archivo llamado "saludos.json" dentro de nuestra subcarpeta "idiomas":

```
{
  "es" : "Hola",
  "en" : "Hello"
}
```

Después, podemos modificar el contenido de los archivos "es.js" y "en.js" para que no pongan literalmente el texto, sino que lo cojan del archivo JSON, incluyéndolo. Nos quedarían así:

- Archivo "es.js":

```
const textos = require('./saludos.json');

module.exports = {
  saludo : textos.es
};
```

- Archivo "en.js":

```
const textos = require('./saludos.json');

module.exports = {
  saludo : textos.en
};
```

La forma de acceder a los textos desde el programa principal no cambia, sólo lo ha hecho la forma de almacenarlos, que queda centralizada en un archivo JSON, en lugar de en múltiples archivos Javascript. De este modo, ante cualquier errata o actualización, sólo tenemos que modificar el texto en el archivo JSON y no ir buscando archivo por archivo. Además, nos evita el problema de las *magic strings* (cadenas que los programadores ponen a mano donde toca, suponiendo que están bien escritas y que no van a hacer falta desde otra parte de la aplicación).

3.3. Incluir módulos con "import"

Desde la especificación ES6, como hemos comentado, Javascript admite la modularización de sus ficheros fuente. Sin embargo, dicha modularización no emplea la instrucción `require` fuera de Node.js, sino que generalmente se emplea la instrucción `import`, que tiene una sintaxis como esta (ponemos como ejemplo la librería `fs` empleada anteriormente):

```
import * as fs from 'fs'
```

En algunos casos, dependiendo de cómo se exporten los contenidos del módulo en cuestión, también la podremos encontrar abreviada así:

```
import fs from 'fs'
```

En cualquier caso, en estos apuntes emplearemos la nomenclatura tradicional de Node con la instrucción `require`.

4. 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:

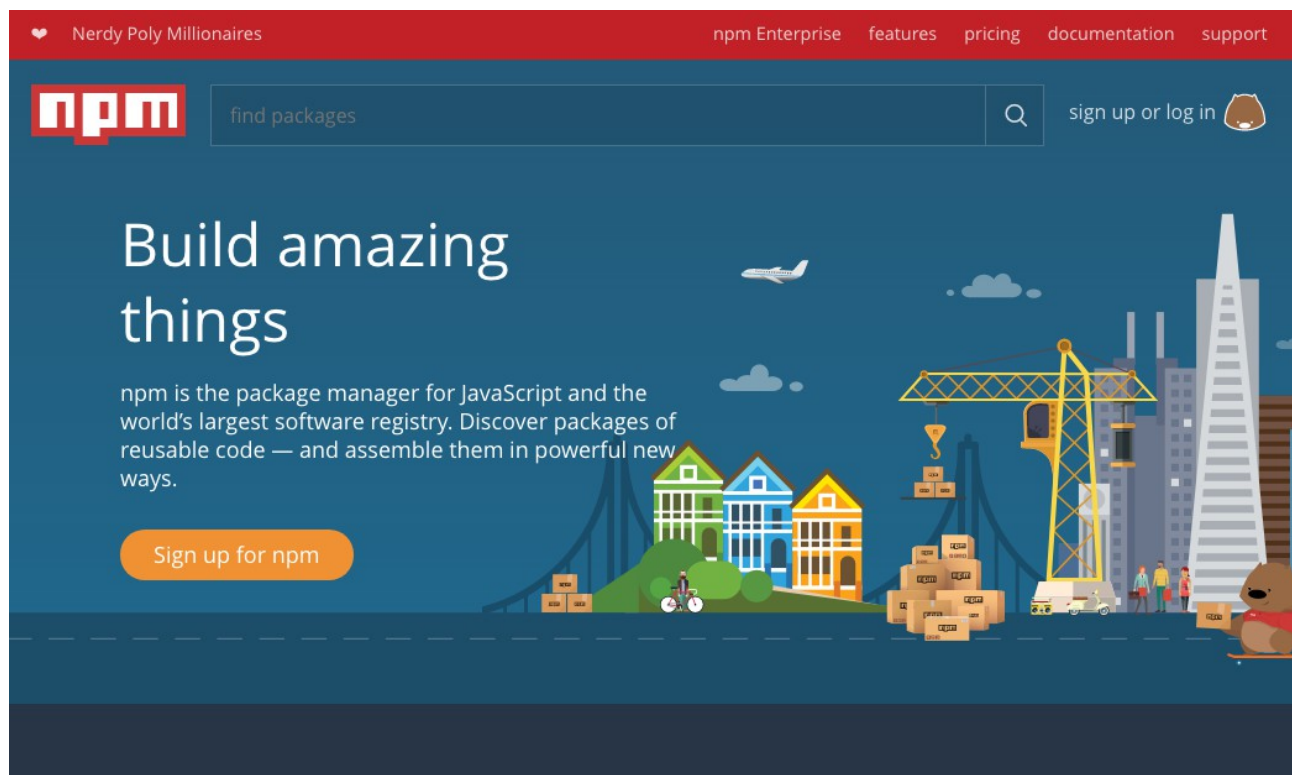
```
npm -v
```

aunque también nos servirá el comando `npm --version`. Para la versión de Node comentada en estos apuntes (8.4.0), la correspondiente versión de npm que se instalará será la 5.3.0.

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, 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 implementar toda la infraestructura necesaria, que podemos descargar e instalar.

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.

The screenshot shows the npm package page for 'express'. At the top, there's a search bar with 'npm' and 'find packages'. Below that, the package name 'express' is displayed with a 'public' tag. The word 'express' is written in a large, stylized font. Below it, a tagline reads 'Fast, unopinionated, minimalist web framework for node.' A row of badges shows 'npm v4.15.4', 'downloads 14M/month', 'linux passing', 'windows passing', and 'coverage 100%'. A code block contains a snippet of JavaScript code for setting up an Express app. To the right, there's a section titled 'Use this within your firewall' with text about private registries. Below that, it says 'npm install express' and 'how? learn more'. Further down, it mentions 'dougwilson published 4 weeks ago', '4.15.4 is the latest of 253 releases', and provides links to 'github.com/expressjs/express' and 'expressjs.com'. At the bottom right, it shows the MIT license and a list of collaborators.

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.

4.1. 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 "Pruebas", cuya carpeta podemos crear ya.

4.1.1. 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, 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 si más.

```
Press ^C at any time to quit.  
package name: (pruebanpm)  
version: (1.0.0)  
description:  
entry point: (index.js)  
test command:  
git repository:  
keywords:  
author: █
```

Observad que el nombre del proyecto lo toma automáticamente de la carpeta en que se encuentra, y el archivo principal de la aplicación suele llamarse "index.js" (o "app.js" en algunos ejemplos que podéis encontrar por Internet).

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.

4.1.2. 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 --save 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:

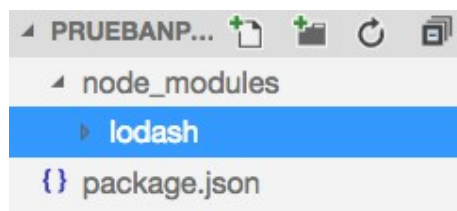
```
npm install --save nombre_modulo@1.1.0
```

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 --save 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:



- El flag `--save` sirve para que, además de descargar el módulo, se modifique el archivo "package.json" de configuración con el nuevo módulo incluido en el bloque de dependencias:

```
{  
  "name": "pruebanpm",  
  ...  
}
```

```

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

```

- 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 de momento.

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, este código 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, 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 ]

```

Ejercicio 3

Crea una carpeta llamada "Tema1_EnlazarLista" en tu espacio de trabajo, en la carpeta "Ejercicios". Dentro, crea un archivo "package.json" utilizando el comando `npm init` visto antes. Deja los valores por defecto que te plantea el asistente, y pon tu nombre como autor.

Después, instala el paquete "lodash" como se ha explicado en el ejemplo anterior, y consulta su documentación ([aquí](#)), para hacer un programa en un archivo "index.js" que, dado un vector de nombres de personas, los muestre por pantalla separados por comas. Deberás definir a mano el array de nombres dentro del código. Por ejemplo, para el array ["Alex", "Arturo", "Julio", "Nacho"], la salida del programa deberá ser:

```
Alex,Arturo,Julio,Nacho
```

NOTA: revisa el método `join` dentro de la documentación de "lodash", puede serte muy útil para este ejercicio.

4.1.3. 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": "^1.1.2" 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 --save
```

4.1.4. 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 el módulo "express":

```
{
  ...
  "dependencies": {
    "lodash": "^4.17.4",
    "express": "*"
  }
}
```

Para hacer efectiva la instalación de los módulos que añadimos a mano en 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.

4.1.5. Desinstalar un módulo

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

```
npm uninstall --save nombre_modulo
```

Nuevamente, el flag `--save` se emplea para actualizar los cambios en el archivo "package.json".

4.1.6. 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 "node_modules" interna al propio módulo.

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.

4.2. 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. 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).

4.2.1. 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 *root*):

```
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.

Ejercicio 4

Instala el módulo "nodemon" a nivel global en tu sistema, y ejecuta con él el ejercicio anterior. Prueba a modificar los nombres del vector y comprueba cómo se actualiza automáticamente la salida por pantalla.

4.2.2. 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
```

4.3. Orden de inclusión de los módulos

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');
```


5. Depuración de código

Existen diferentes alternativas para depurar el código de nuestras aplicaciones Node.js. En esta sección veremos tres:

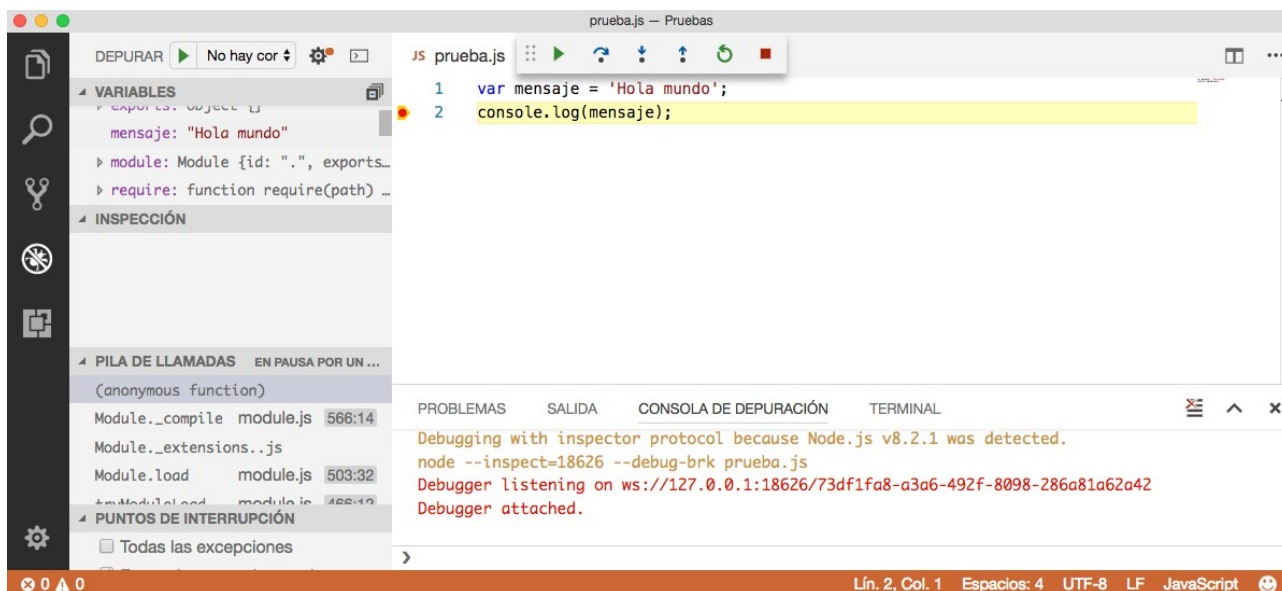
- Utilizando nuestro propio IDE (Visual Studio Code)
- Desde terminal
- Utilizando Google Chrome


En cualquiera de los tres casos, hay que decir que desde Node versión 8 se ha estabilizado bastante la depuración de código, en cuanto a opciones incorporadas que utilizar. Nos valdremos de estas herramientas directa o indirectamente a la hora de depurar nuestro código.

5.1. Depurar desde Visual Studio Code

Visual Studio Code ofrece un depurador para nuestras aplicaciones Node. Para entrar en modo depuración, hacemos clic en el icono de depuración del panel izquierdo (el que tiene forma de *bug* o chinche ).

Podemos establecer *breakpoints* en nuestro código haciendo clic en la línea en cuestión, en el margen izquierdo (como en muchos otros editores de código). Después, podemos iniciar la depuración con F5, o bien con el menú *Depurar > Iniciar depuración*, o con el icono de la flecha verde de *play* de la barra superior. Al llegar a un *breakpoint*, podemos analizar en el panel izquierdo los valores de las variables y el estado de la aplicación.



También podemos continuar hasta el siguiente *breakpoint* (botón de flecha verde), o ejecutar paso a paso con el resto de botones de la barra de depuración. Para finalizar la depuración, hacemos clic en el icono del cuadrado rojo de *stop* (si no se ha detenido ya la aplicación), y volvemos al modo de edición haciendo clic en el botón de explorador del panel izquierdo .

La "consola de depuración" que aparece en el terminal inferior realmente es un terminal REPL (*Read Eval Print Loop*), lo que significa que desde ella podemos acceder a los elementos de nuestro programa (variables, objetos, funciones) y obtener su valor o llamarlos. Por ejemplo, en el caso anterior, podríamos teclear "mensaje" en el terminal y ver cuánto vale esa variable:

```
PROBLEMAS  SALIDA  CONSOLA DE DEPURACIÓN  TERMINAL  ≡ ^ x
Debugger listening on ws://127.0.0.1:18626/73df1fa8-a3a6-492f-8098-286a81a62a42
Debugger attached.

mensaje
"Hola mundo"
> |
```

5.2. Depurar desde terminal

En el caso de que no utilicemos Visual Studio Code, o que prefiramos utilizar el terminal, podemos lanzar nuestra aplicación en modo depuración con el siguiente comando:

```
node inspect archivo.js
```

Una vez hecho esto, se inicia el modo de depuración. En el caso de nuestro "Hola mundo" anterior, el terminal quedará más o menos así:

```
Mac-Mini-de-Ignacio:Pruebas nacho$ node inspect prueba.js
< Debugger listening on ws://127.0.0.1:9229/1f0b204a-fbee-4306-bc1e-65aebd380be6
< For help see https://nodejs.org/en/docs/inspector
< Debugger attached.
Break on start in prueba.js:1
> 1 (function (exports, require, module, __filename, __dirname) { var mensaje =
'Hola mundo';
  2 console.log(mensaje);
  3 });
debug> █
```

El símbolo inferior indica que el terminal está esperando comandos de depuración. Los más elementales son los siguientes:

- `list(nlineas)`, donde `nlineas` será un número entero, mostrará en el terminal las `nlineas` líneas de código anteriores y posteriores al punto en el que estamos
- `n` pasará a ejecutar la siguiente instrucción (ejecución paso a paso)
- `c` ejecutará el programa hasta su finalización, o hasta encontrar un punto de ruptura (*breakpoint*). Para definir puntos de ruptura en el código, se puede hacer añadiendo la instrucción `debugger`; donde queramos poner el punto de ruptura. Por ejemplo:

```
var mensaje = 'Hola mundo';
debugger;
console.log(mensaje);
```

De este modo, el depurador se detendrá en esa línea al ejecutar el comando `c`.

- `repl` inicia un terminal REPL, como el que hemos visto para Visual Studio Code. El símbolo de prompt cambiará, y podremos comprobar el valor de variables u objetos, o llamar a funciones:

```
debug> repl
Press Ctrl + C to leave debug repl
> mensaje
'Hola mundo'
> █
```

Para salir de dicho terminal REPL, pulsaremos Control+C, y volveremos al terminal de depuración (debug>)

- Desde el modo de depuración normal (debug>), si queremos finalizar la depuración pulsaremos dos veces Control+C.

5.3. Depurar desde Google Chrome

Si tenemos disponible Google Chrome, podemos valernos de las herramientas para desarrolladores que incorpora (*Developer Tools*) para utilizar el depurador. Para ello, desde un terminal normal ejecutaremos este comando:

```
node --inspect-brk archivo.js
```

Notar que es similar al que escribíamos para depurar por terminal, modificando el parámetro *inspect*. Una vez hecho esto, la aplicación queda a la espera de ser depurada. Si abrimos Chrome y accedemos a la URL *chrome://inspect*, podemos acceder al depurador desde el enlace *Open dedicated DevTools for Node*.



Se abrirá una ventana con el depurador. Desde la pestaña *Sources* podemos examinar el código fuente del programa:



Además, podemos establecer *breakpoints* de forma visual haciendo clic izquierdo sobre el número de línea (se marcará en azul, como en la imagen anterior), y ejecutar la aplicación de forma continuada o paso a paso con los controles de la parte superior derecha. Finalmente, si abrimos la consola inferior de esta pestaña *Sources*, accedemos a un terminal REPL como los vistos anteriormente, para examinar el valor de variables, objetos, o llamar a funciones. Notar cómo examinamos la variable *mensaje* en el ejemplo anterior.

5.4. Depurar con "nodemon"

Si hemos instalado el módulo nodemon de forma global en nuestro sistema, podemos lanzar los comandos anteriores utilizando nodemon en lugar de node.

```
nodemon inspect archivo.js
```

```
nodemon --inspect-brk archivo.js
```

De esta forma, cada vez que realicemos un cambio en el código se reiniciará la depuración automáticamente, lo que puede resultar bastante cómodo.

Ejercicio 5

Ve a la carpeta "PruebasRequire" y utiliza el depurador en cualquiera de las tres formas explicadas (Visual Studio Code, terminal o Google Chrome). Añade un punto de ruptura dentro de cada función del archivo "utilidades.js" (funciones sumar y restar) para que, cuando se les llame, podamos examinar el valor de los parámetros que reciben como entrada.

6. Sobre require o module.exports

Quizá algunos de vosotros os habréis preguntado... ¿cómo es que puedo tener accesibles variables o métodos que yo no he definido al empezar, como `require`, o `module.exports`? La respuesta os ha pasado antes por delante, aunque puede que desapercibida.

Si habéis probado a depurar algún programa desde terminal con `node inspect`, aparecen una serie de líneas. Muchas de ellas son de información del depurador, pero después hay una que muestra una función que encapsula nuestro código:

```
Mac-Mini-de-Ignacio:Pruebas nacho$ node inspect prueba.js
```

```
< Debugger listening on ws://127.0.0.1:9229/949....
```

```
< For help see https://nodejs.org/en/docs/inspector
```

```
< Debugger attached.
```

```
Break on start in prueba.js:1
```

```
> 1 (function (exports, require, module, __filename, __dirname) { var mensaje =  
  'Hola mundo';  
  2 console.log(mensaje);  
  3 });
```

```
debug>
```

¿Qué significa esto? Pues que, cuando se compila y ejecuta nuestro código, Node.js lo encapsula dentro de una función, y le pasa como parámetros los elementos externos que puede necesitar, como por ejemplo `require`, o `module` (en cuyo interior encontraremos `exports`).

6.1. `exports` y `module.exports`

Los más avisados habréis advertido que, en la función con la que Node.js encapsula nuestro código, existe un parámetro `exports`. Entonces...

- ¿Hay un `exports` por un lado y un `module.exports` por otro? La respuesta es que SI
- ¿Existe diferencia entre ambos? La respuesta también es que SI. A priori, ambos elementos apuntan al mismo objeto en memoria, es decir, `exports` es un atajo para no tener que escribir `module.exports`. Pero...
 - Si cometemos el error de reasignar la variable (por ejemplo, haciendo `exports = a`), entonces las referencias dejan de ser iguales.
 - Por otra parte, si queréis ver el código fuente de la función `require`, veréis que lo que devuelve es `module.exports`, por lo que, en caso de reasignar la variable `exports`, no nos serviría de nada.

La moraleja de todo esto es que, en principio, `exports` y `module.exports` sirven para lo mismo siempre que no las reasignemos. Pero durante todo este módulo seguiremos nuestro propio consejo: usaremos siempre `module.exports` para evitar problemas.