

# Node.js

---

*en el desarrollo de aplicaciones web y servicios*

## 1. Introducción a Node.js

Ignacio Iborra Baeza

# Índice de contenidos

<b>Node.js</b>	<b>1</b>
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
1.2.1. Requisitos para que Javascript pueda correr en el servidor	4
1.2.2. Consecuencia: un único lenguaje de desarrollo web	4
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
2.1.1. Instalación en Windows y Mac	6
2.1.2. Instalación en Linux o en la máquina virtual	7
2.1.3. Resultado de la instalación	8
2.1.4. Actualizar desde una versión previa	8
2.2. <i>Primeras pruebas con el comando "node"</i>	8
2.2.1. Nuestro primer programa Node	8
2.2.2. Uso de "node" como intérprete de comandos	9
2.2.3. Similitudes entre este terminal y la consola de Chrome	9
2.3. <i>Integrar Node.js en Visual Studio Code</i>	10
2.3.1. Preparar el espacio de trabajo	10
2.3.2. Crear y editar un proyecto básico	10
2.3.3. Ejecutar un archivo Node desde Visual Studio Code	11
3. Depuración de código	12
3.1. <i>Depurar desde Visual Studio Code</i>	12
3.2. <i>Depurar desde terminal</i>	13
3.3. <i>Depurar desde Google Chrome</i>	14
4. Ejercicios	15
4.1. <i>Ejercicio 1</i>	15
4.2. <i>Ejercicio 2</i>	15

# 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 esta sesión 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 varias 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. Una siguiente etapa, vinculada a la anterior, tuvo lugar con la aparición de distintos frameworks Javascript para desarrollo de aplicaciones web en el lado del cliente, o *frontends*. Mediante una serie de funcionalidades incorporadas, y de librerías externas, permiten dotar a la aplicación cliente de una estructura muy determinada con unos añadidos que facilitan, entre otras cosas, la compartición de variables entre vistas o páginas, o la generación dinámica de contenido HTML en la propia vista. Hablamos, fundamentalmente, de frameworks como Angular o React.
4. La última etapa ha llegado con la expansión de Javascript al lado del servidor. Hasta este momento sólo se utilizaba en la parte cliente, es decir, fundamentalmente 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.

### 1.2.1. Requisitos para que Javascript pueda correr en el servidor

¿Qué características tienen lenguajes como PHP, ASP.NET 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 enviar 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. Sin embargo, la API asíncrona y 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

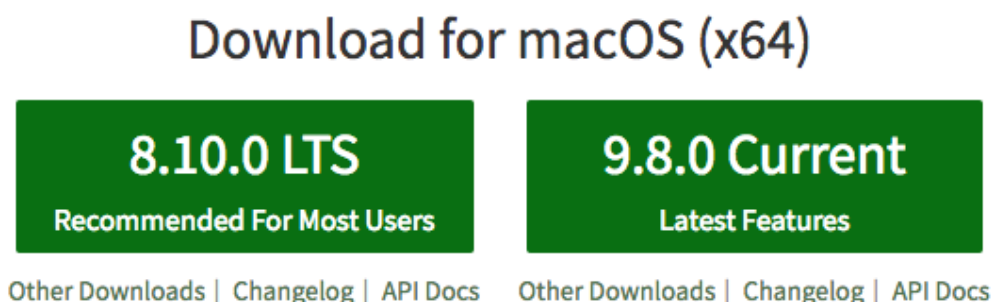
---

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](https://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. Para el cometido de este curso es indiferente qué versión instalar, pero dado que ya se ha dado el salto a la versión 9, probaremos con esta última.

#### 2.1.1. Instalación en Windows y Mac

En el caso de querer instalar Node sobre un sistema Windows o Mac, hacemos clic 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.



### 2.1.2. Instalación en Linux o en la máquina virtual

Si estamos utilizando un sistema Linux, o la máquina virtual proporcionada para este curso, se recomienda instalar Node.js desde un repositorio. Vamos a suponer que utilizamos una distribución Debian (o Ubuntu, o similares), como la de la máquina virtual que tenéis disponible. En ese caso, podemos escribir estos comandos en un terminal en modo *root*, es decir, anteponiendo el comando *su*, con la correspondiente contraseña de *root*:

```
apt-get update
apt-get upgrade
apt-get install curl
curl -sL https://deb.nodesource.com/setup_9.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.

### 2.1.3. Resultado de la instalación

Tras la instalación en cualquiera de los sistemas operativos comentados, tendremos disponible 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.

### 2.1.4. 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 (`setup_9.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 curso.

## 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:

```
v9.8.0
```

**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.

Llegados a este punto, ya puedes realizar el [Ejercicio 1](#) de los propuestos al final de la sesión.

### 2.2.1. Nuestro primer programa Node

El 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
```



### 2.2.2. Uso de "node" como intérprete de comandos

Si escribimos el comando `node` a secas en el terminal, aparecerá un "prompt" con una flecha hacia 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:

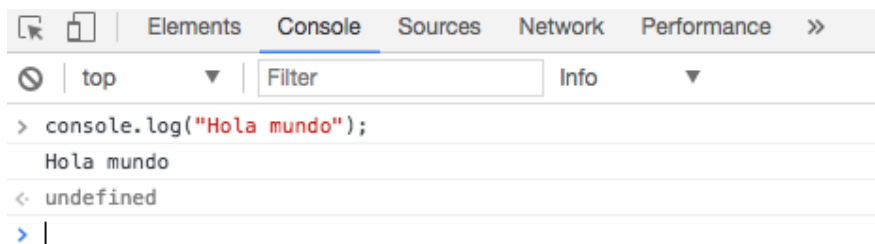
```
> console.log("Hola mundo");
```

```
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)`.

### 2.2.3. 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*. Esta opción 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 es uno de los muchos editores que podemos emplear para desarrollar aplicaciones Node. Disponéis de una versión ya instalada en la máquina virtual que se os proporciona, y si lo queréis instalar en vuestro sistema, la instalación no ofrece ninguna complicación. Podéis descargarlo desde su [web oficial](#).

Este IDE ofrece una integración muy interesante con Node.js, de manera que podemos editar, ejecutar y depurar nuestras aplicaciones desde el propio 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, y para centralizar de alguna forma todo el trabajo del curso, 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 dos subcarpetas:

- **Pruebas**, donde guardaremos todos los proyectos de prueba y ejemplo que hagamos
- **Ejercicios**, donde almacenaremos los ejercicios propuestos de cada sesión

La estructura de carpetas quedará entonces como sigue:

- ProyectosNode
  - Pruebas
  - Ejercicios

### 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. También podemos abrir la carpeta arrastrándola desde algún explorador de carpetas hasta una instancia abierta de Visual Studio Code.

De momento la carpeta 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:

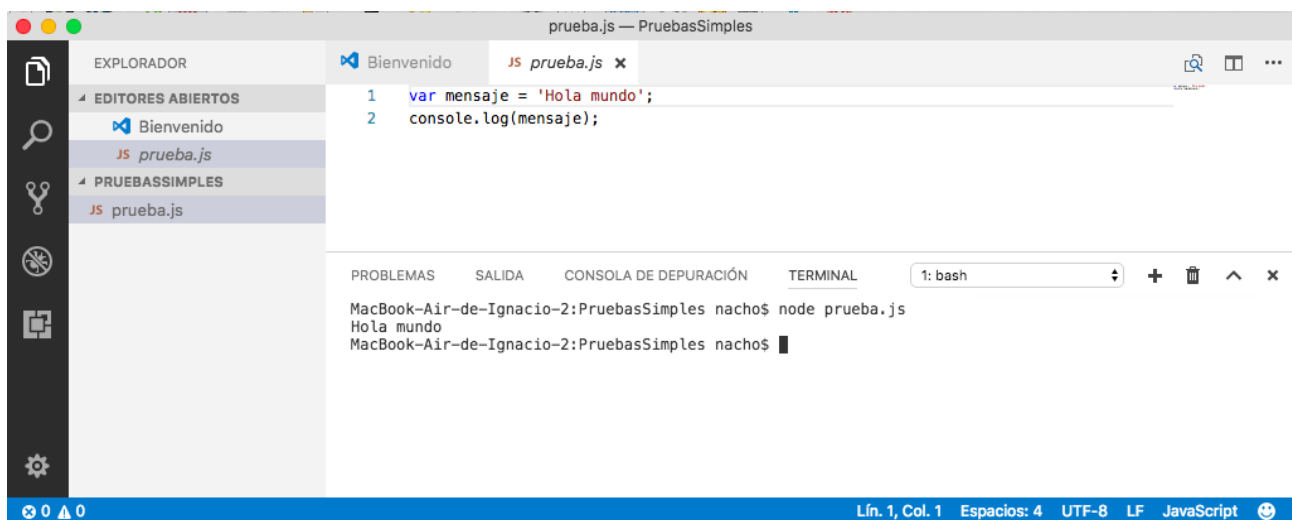


*Crear nuevos archivos y carpetas en la actual*

### 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:




## 3. Depuración de código

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

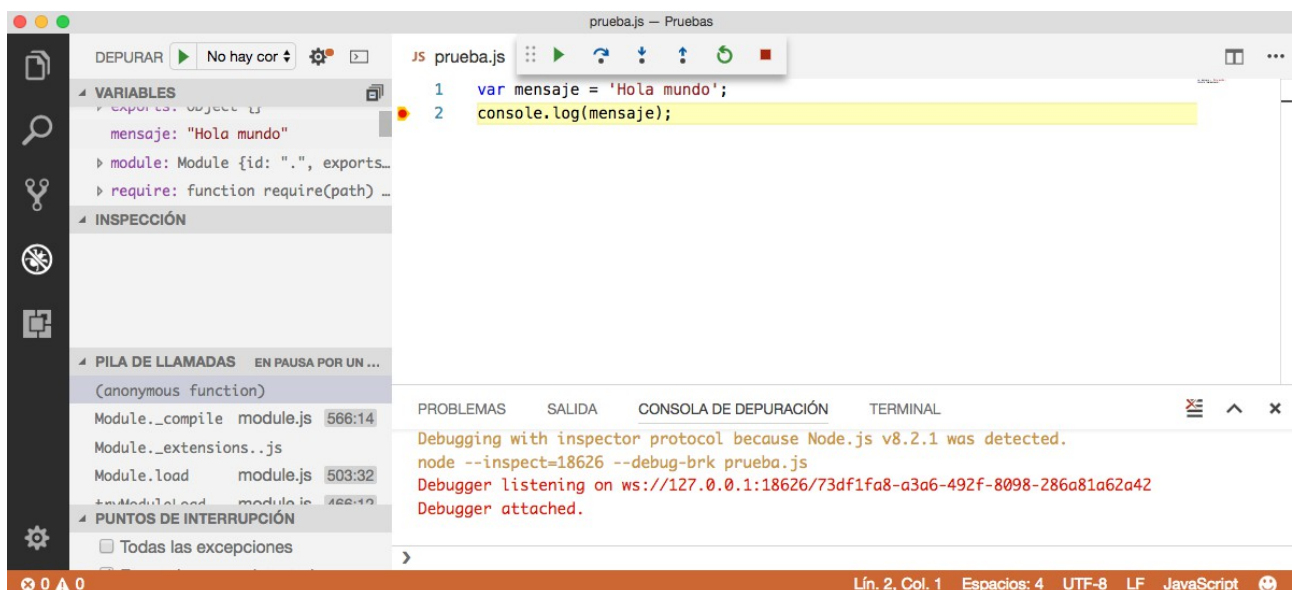
- Utilizando nuestro propio IDE (Visual Studio Code)
- Desde terminal
- Utilizando Google Chrome


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.

### 3.1. 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
Debugger listening on ws://127.0.0.1:18626/73df1fa8-a3a6-492f-8098-286a81a62a42
Debugger attached.

mensaje
"Hola mundo"
>|
```

## 3.2. 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`. Esta forma de establecer *breakpoints* también es válida para la depuración desde Visual Studio Code vista antes, en lugar de hacer clic sobre el margen izquierdo del editor de código.

- `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.

### 3.3. Depurar desde Google Chrome

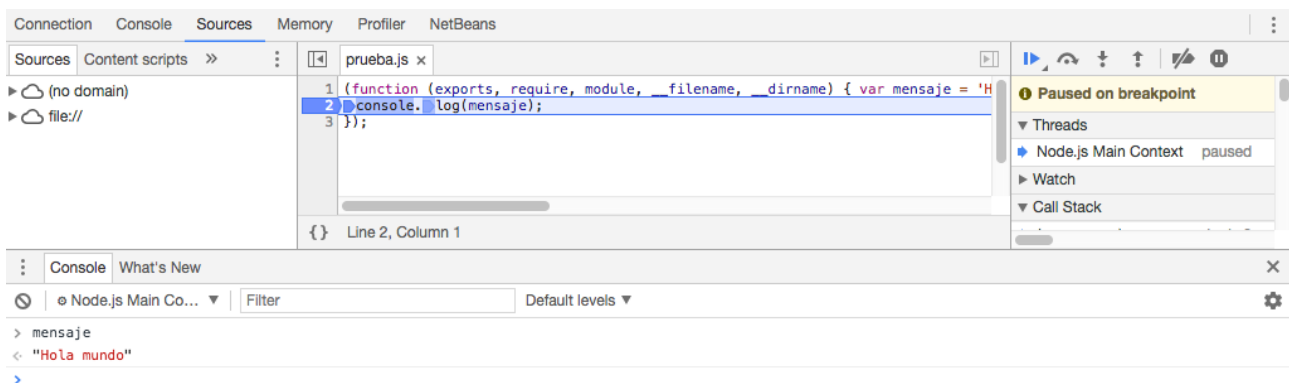
Si tenemos disponible Google Chrome, podemos valernos de las herramientas para desarrolladores que incorpora (*Developer Tools*) para utilizar el depurador. Para ello, desde un terminal normal ejecutaremos este comando:

```
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), o con la instrucción `debugger`; vista antes para el terminal. Después, podemos 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.

En este punto, puedes intentar realizar el [Ejercicio 2](#) de los propuestos al final de esta sesión.

## 4. Ejercicios

---

Si has seguido los pasos explicados en estos apuntes, deberías tener una carpeta llamada "ProyectosNode" en tu espacio de trabajo o carpeta personal, y dentro las carpetas "Pruebas" y "Ejercicios". Si no es así, crea estas carpetas antes de seguir, y una vez creadas, crea una subcarpeta llamada "Sesion1" dentro de tu carpeta "ProyectosNode/Ejercicios".

### 4.1. Ejercicio 1

---

Instala Node.js en la máquina virtual proporcionada (o en tu propio sistema operativo, si lo prefieres), siguiendo los pasos indicados en estos apuntes. Tras la instalación, ejecuta el comando `node -v` para que muestre la versión que tienes instalada. Adjunta una captura de pantalla del terminal con el comando y el resultado, en una imagen llamada *prueba\_node* con la extensión que quieras (JPG, PNG...).

### 4.2. Ejercicio 2

---

Crea una carpeta llamada "Ejercicio\_1\_2" en la carpeta "ProyectosNode/Ejercicios/Sesion1", ábrela con Visual Studio Code y crea un archivo llamado "depuracion.js". Dentro, introduce este código y guarda el archivo:

```
1  var m = 1, n = 2;
2
3  for(i = 1; i <= 5; i++) {
4      m = m * i;
5      n = n + m * n;
6  }
7
8  console.log(n);
```

Utiliza el depurador de cualquiera de las formas explicadas (a través de Visual Studio Code, terminal o con Google Chrome) para averiguar el valor de la variable `n` tras ejecutarse la línea 6 de código (pon un *breakpoint* en la línea 7, por ejemplo). Haz una captura de pantalla mostrando lo que haces para averiguar el valor, y el resultado que te muestra. Adjunta dicha captura con el nombre *depuracion*, y la extensión que prefieras (JPG, PNG...).