

Tema 1

Node.js

Micael Gallego

micael.gallego@gmail.com
@micael_gallego

Pablo Fuente

pablofuenteperez@gmail.com
@fuentepab

Introducción a Node.js



¿Qué es Node.js?

- Es un **runtime** que permite ejecutar **JavaScript** fuera de un navegador web
- Está basado en **V8**, la máquina virtual de Google Chrome (<https://v8.dev>)
- Como se programa en **JavaScript**, el modelo de programación es **asíncrono** ya que las llamadas de I/O no son bloqueantes, lo que le hace muy **escalable** para **aplicaciones de red**

<https://nodejs.org>

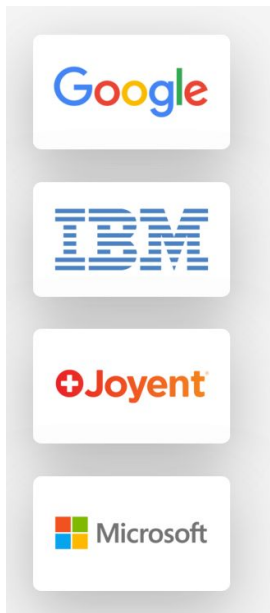
Introducción a Node.js

Soporte en la industria

Node.js está desarrollado bajo el paraguas de



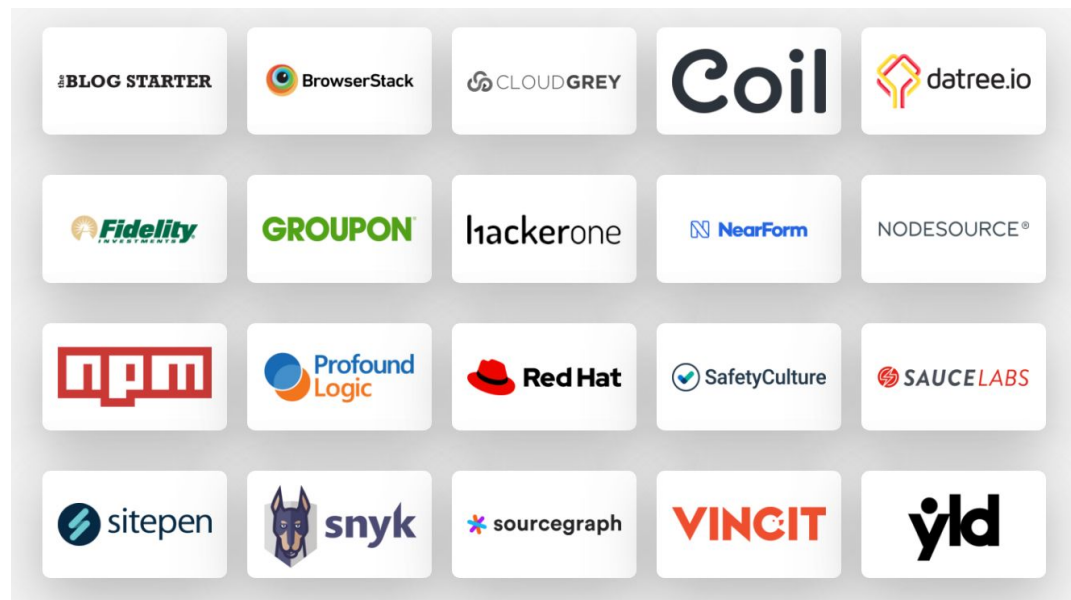
Premium



Gold



Silver



Plataforma con mucha adopción

- Tiene una **gran comunidad de desarrolladores**
- Está soportado de forma oficial en los **proveedores cloud** (como Java, Python...)
- Tiene una **gran cantidad de paquetes NPM**
- Cada vez se usa más a nivel **empresarial**
- Incorpora las últimas versiones del estándar **EcmaScript** (lenguaje y librerías muy completas)

Tipos de aplicaciones

Con Node.js se pueden desarrollar todo tipo de aplicaciones:

- Servicios de red (Aplicaciones web MVC, APIs REST)
- Herramientas por línea de comandos (CLI)
- Aplicaciones web SPA
- Aplicaciones con interfaz gráfico de usuario

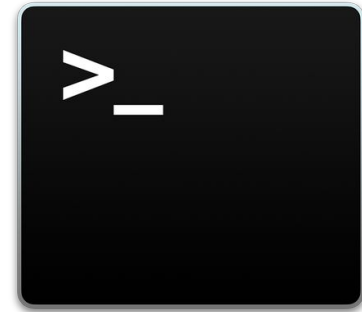
Servicios de red

- Aplicaciones web MVC:
Generación de HTML en servidor
- Backend para SPA
- Protocolos de comunicación: API REST, WebSockets, AMQP, gRPC...
- Acceso a base de datos: Relacionales, NoSQL...
- Serverless, Microservicios...



Herramientas por línea de comandos

- Herramientas para desarrollo:
 - Linters
 - Compiladores
 - Testing de rendimiento
- Gestión de proyectos: angular-cli, vue-cli...
- Scripts de sistemas



Aplicaciones Web SPA

- Se ejecutan en el browser
- El código Node.js tiene que adaptarse porque no se puede ejecutar directamente en el navegador
- Para ello se usan "bundlers"



webpack



rollup.js

Aplicaciones con interfaz gráfico de usuario

Se combina en una aplicación **Node.js** (para acceso al sistema) con un navegador web (**Chromium**) para el interface de usuario



<https://electronjs.org/>



<https://nwjs.io/>

Librerías

- En **Node.js** no se pueden usar librerías del browser como **DOM**, **BOM**, etc. porque no hay interfaz de usuario
- Node.js ofrece por defecto unas **mínimas librerías** para interactuar con el **sistema operativo**
- Existen muchas librerías libres disponibles en NPM, el gestor de paquetes oficial de Node



<https://npmjs.com>

Librerías

Las librerías incluidas por defecto en Node.js ofrecen las siguientes funcionalidades:

- Gestión de procesos
- Gestión de datos en memoria nativa del sistema
- Acceso a la línea de comandos
- Sistema de ficheros
- Redes (Http, Sockets)
- Flujos de bytes

<https://nodejs.org/api>

Versiones

Node.js ofrece 2 tipos de versiones

- Activa LTS (*Long Term Support*): Soporte extendido
- Actual (*Current*): últimas funcionalidades

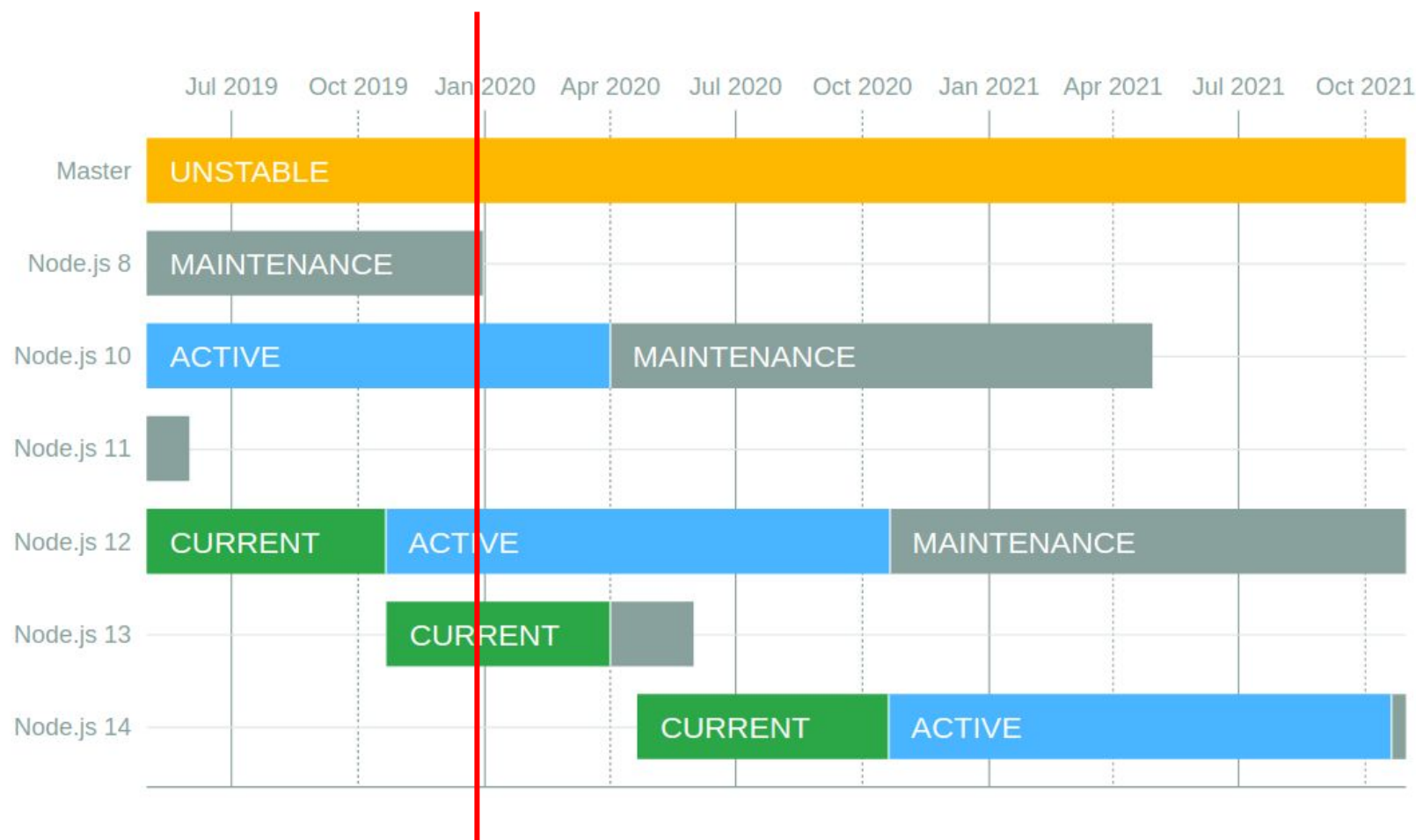
En diciembre de 2019 las versiones son

- Activa LTS: 12.13.1 LTS
- Actual: 13.1.0 Current

<https://github.com/nodejs/LTS#lts-schedule1>

Introducción a Node.js

Versiones



Rendimiento

- El código **JavaScript** no se puede ejecutar de forma tan eficiente como Java o C# debido a su naturaleza dinámica
- Su modelo **asíncrono**, ofrece una escalabilidad igual o superior a aplicaciones Java equivalentes que no sean asíncronas
- El **modelo de programación es mucho más sencillo** al no existir varios hilos de ejecución (no hay condiciones de carrera)
- Se considera una **solución aceptable en servicios de red** que con mucha I/O y poca algoritmia

Introducción a Node.js

Instalación

LTS Recommended For Most Users	Current Latest Features	
 Windows Installer <small>node-v12.13.1-x86.msi</small>	 macOS Installer <small>node-v12.13.1.pkg</small>	 Source Code <small>node-v12.13.1.tar.gz</small>

Windows Installer (.msi)

Windows Binary (.zip)

macOS Installer (.pkg)

macOS Binary (.tar.gz)

Linux Binaries (x64)

Linux Binaries (ARM)

Source Code

32-bit	64-bit
32-bit	64-bit
64-bit	
64-bit	
64-bit	
ARMv7	ARMv8
node-v12.13.1.tar.gz	

<https://nodejs.org/en/download>

Instalación

En Unix con esto es suficiente:

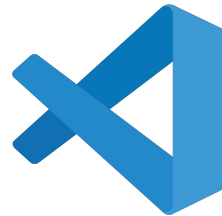
```
curl -sL https://deb.nodesource.com/setup_12.x | sudo -E bash -  
sudo apt-get install nodejs
```

IDE

Algunos IDEs vienen con Node.js integrado, pero se recomienda instalar Node en el sistema:



Sublime Text



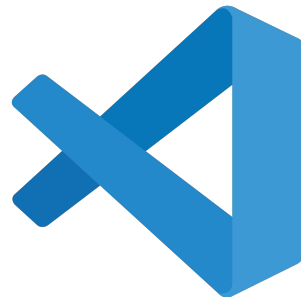
Visual Studio
Code



WebStorm



- Usaremos **Visual Studio Code** para desarrollar y depurar
- Usaremos algunas **APIs** que vienen incluidas en Node.js
- Usaremos **NPM** para instalar librerías externas



Hello world!

ejem1

- Crear una carpeta
- Crear un fichero **app.js**

```
console.log('Hello world!');
```

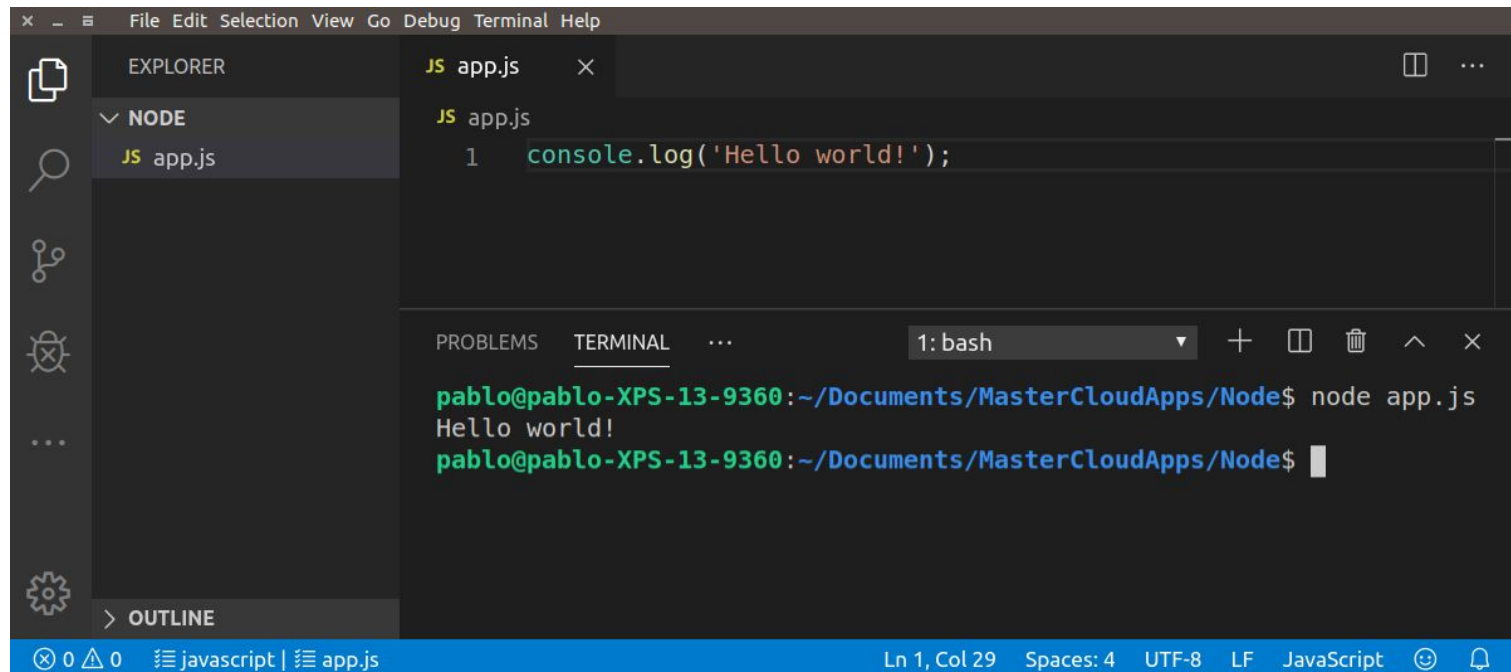
- Ejecutar el comando

```
$ node app.js
```

Aplicaciones de consola

Visual Studio Code

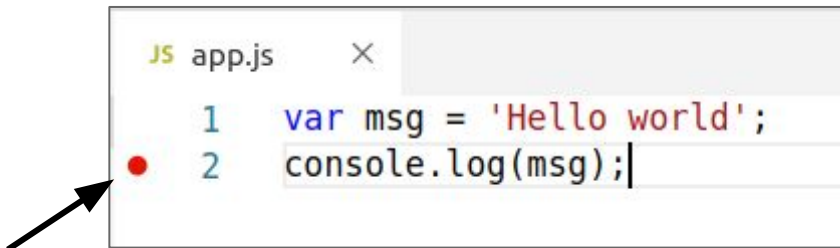
- Abrimos la carpeta con todo el código
- Ejecutamos comandos en el terminal integrado (**Ctrl+`**)



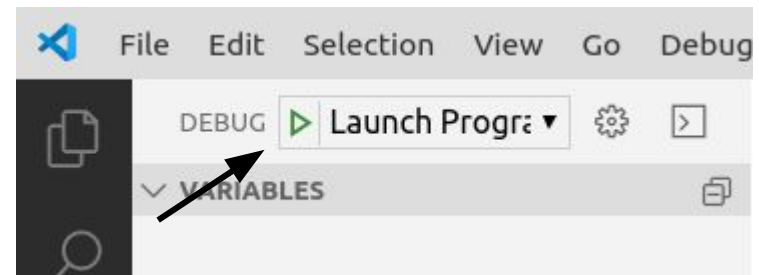
The screenshot displays the Visual Studio Code editor interface. On the left, the Explorer sidebar shows a project named 'NODE' containing a file 'app.js'. The main editor area shows the content of 'app.js', which is a single line of JavaScript code: `console.log('Hello world!');`. Below the editor, the integrated terminal is open, showing the command `node app.js` being executed. The output of the command is 'Hello world!'. The terminal prompt indicates the user is 'pablo' on a machine named 'pablo-XPS-13-9360' in the directory `~/Documents/MasterCloudApps/Node`. The status bar at the bottom shows the file is 'app.js' and the language is 'javascript'.

Depuración en Visual Studio Code

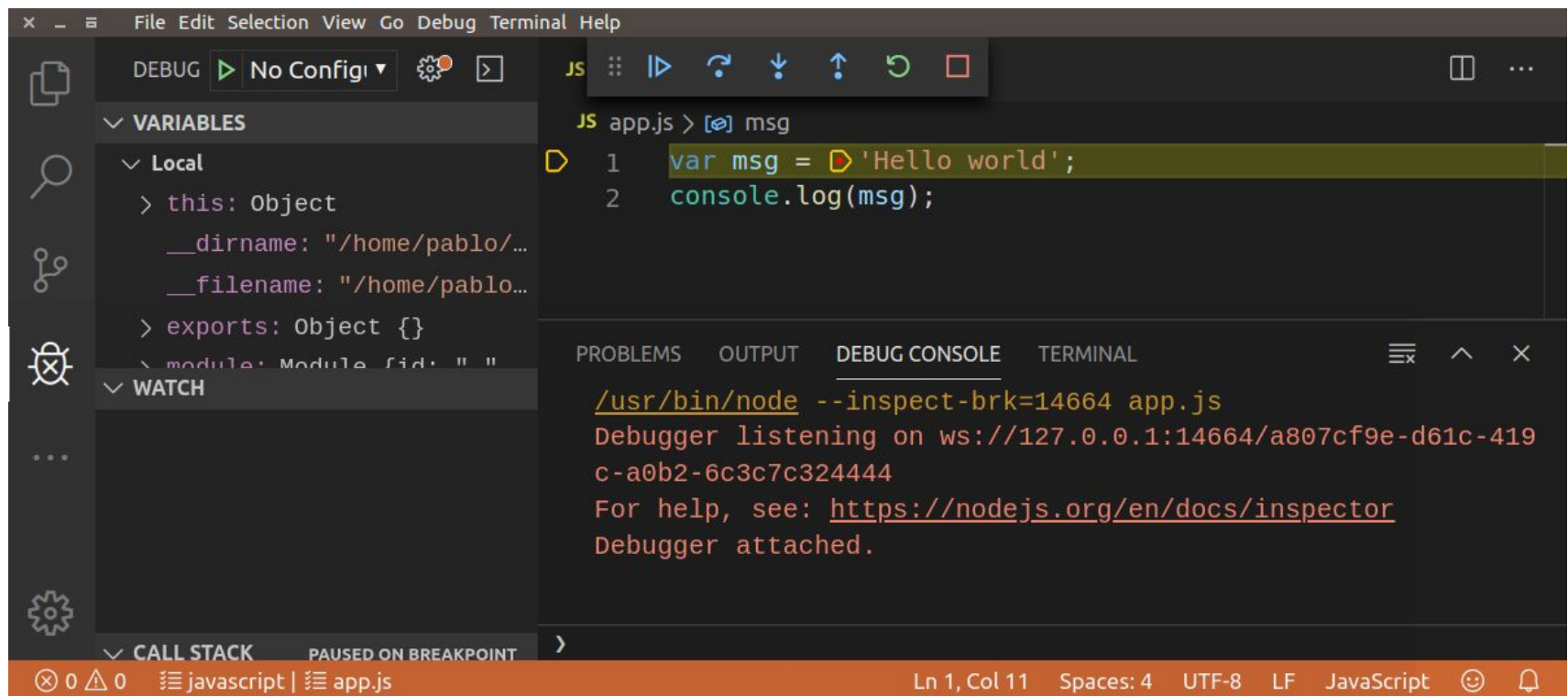
- Para poner un punto de ruptura se marca en la barra a la izquierda del código



- Seleccionamos la vista de debug y pulsamos en "Launch Program"

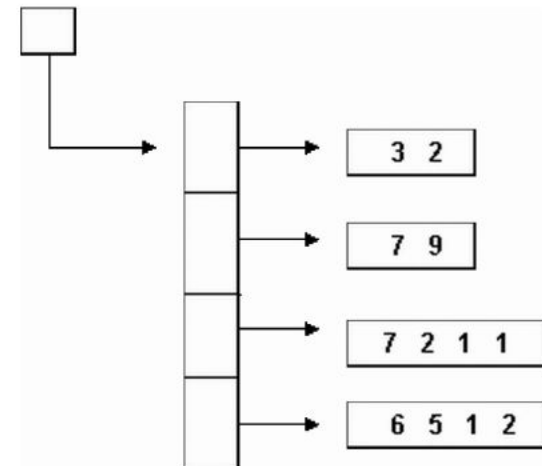
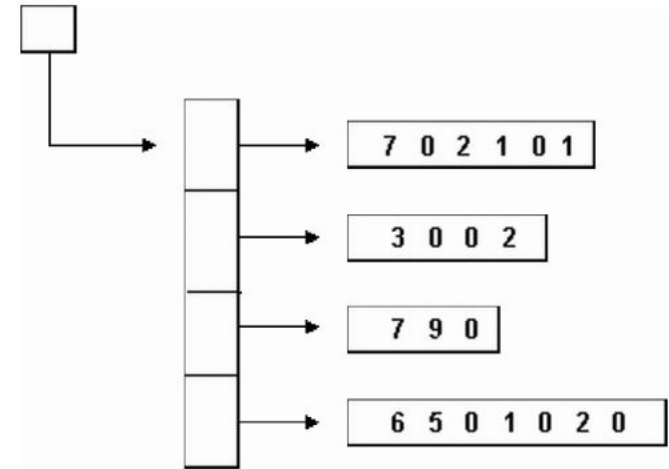


Depuración en Visual Studio Code



Ejercicio 1

- Crear una función que **reciba un array bidimensional, le quite los ceros y ordene las filas de menor a mayor longitud**
- Para probar la función se implementarán varias llamadas con **diferentes arrays** y el resultado se mostrará en la **consola del navegador**



Tipos de Módulos en Node.js

CommonsJS

ES Modules

(experimental)

<https://medium.com/@nodejs/announcing-core-node-js-support-for-ecmascript-modules-c5d6dc29b663>

Tipos de Módulos en Node.js

- Cuando se desarrolló Node, JavaScript no tenía un sistema estándar para modularizar aplicaciones
- Node diseñó su propio sistema de modularización llamado **CommonJS**
- En **ES6 (2015)** se definió el sistema de módulos estándar llamado **ES Modules**
- Node soporta ES Modules desde **13.2.0 (Nov 2019)**

<https://medium.com/@nodejs/announcing-core-node-js-support-for-ecmascript-modules-c5d6dc29b663>

Módulos Node (CommonJS)

- En Node cada **fichero** es un **módulo**
- Si un módulo A quiere usar variables o funciones de un módulo B:
 - El módulo B tiene que **exportar** lo que quiera hacer público

```
exports = ...
```

```
module.exports = ...
```

- El módulo A tiene que **importar** lo que quiera usar

```
var x = require('./module.js')
```

Módulos Node (CommonJS)

ejem2

- 1) El código del módulo se ejecuta al importarlo (**require**)

hello.js

```
console.log('Hello world');
```

app.js

```
require('./hello.js');
```

app2.js

```
require('./hello');
```

La extensión del fichero se puede omitir

Módulos Node (CommonJS)

ejem2

- 2) Para exportar **una única función anónima (default export)**. Se le da nombre al importarla

bar.js

```
module.exports = function() {  
    console.log('bar!');  
}
```

app.js

```
var bar = require('./bar.js');  
bar();
```

Módulos Node (CommonJS)

ejem2

- 3) Para exportar una función con nombre

fiz.js

```
exports.fiz = function(){  
    console.log('fiz!');  
}
```

app.js

```
var module = require('./fiz.js');  
module.fiz();
```

- 4) Otra forma de exportar una función con nombre

fiz.js

```
function fiz(){  
    console.log('fiz!');  
}  
exports.fiz = fiz;
```

app.js

```
var fiz = require('./fiz.js').fiz;  
fiz();
```

Módulos Node (CommonJS)

ejem2

- 5) Se pueden exportar varios elementos y de diferentes tipos (funciones, objetos, valores, clases...)

utils.js

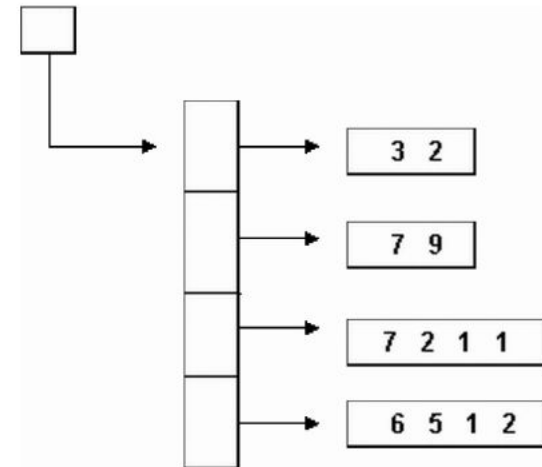
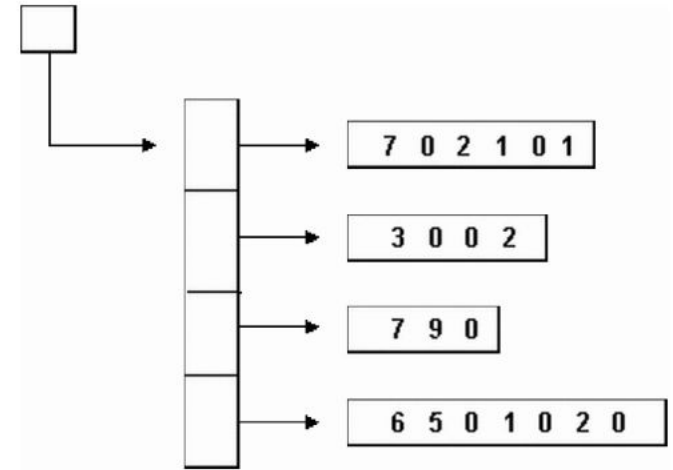
```
var obj = {  
  prop1: 3,  
  prop2: 'value'  
}  
  
function log(msg) {  
  console.log(msg);  
}  
  
class Date {  
  constructor(date) {  
    this.date = date;  
  }  
}  
  
exports.obj = obj;  
exports.log = log;  
exports.Date = Date;
```

app.js

```
var utils = require('./utils.js');  
  
utils.log('message');  
  
console.log(utils.obj);  
  
var date = new utils.Date('01-12-19');
```

Ejercicio 2

- Modifica el Ejercicio 1 para incluir la función de procesamiento de arrays en un módulo
- En el fichero **app.js** se usará ese módulo
- Exporta por **defecto** la función "quitaCeros" desde un módulo
- Exporta con **nombre** la función "ordena" en otro módulo



Librerías incluidas en Node

- Node expone la funcionalidad como **módulos** (CommonJS y ES)
- Se usa **require()** para usar esas librerías en tu módulo
- Existen algunos objetos **globales** que pueden usarse sin **require()**
- El objeto **console** está disponible **sin require**

```
console.log('Hello world!');
```

Librerías incluidas en Node

Lectura de un fichero en Node con el módulo **File System** (fs)

```
var fs = require('fs');

fs.readFile('/home/data.txt', 'utf8', (err, contents) => {
  if (err) {
    return console.error(err);
  }
  console.log(contents);
});

console.log('After calling readFile');
```

Módulos NPM

- Herramienta que permite **descargar** módulos Node de la red
- Existe un **repositorio público** con módulos **software libre**
- También se pueden configurar **repositorios privados** para **módulos privados**



<https://npmjs.com>

Node Package Manager (NPM)

- Una aplicación define los paquetes que necesita en un fichero llamado **package.json**

```
{  
  "name": "my-awesome-package",  
  "version": "1.0.0"  
}
```

- El fichero se puede crear también con el comando

```
$ npm init
```

Node Package Manager (NPM)

- Los paquetes de los que se depende se especifican en la sección "dependencies"

```
{  
  "name": "my-awesome-package",  
  "version": "1.0.0",  
  "dependencies": {  
    "lodash": "4.17.15"  
  }  
}
```

- Para instalar el paquete

```
$ npm install
```

Node Package Manager (NPM)

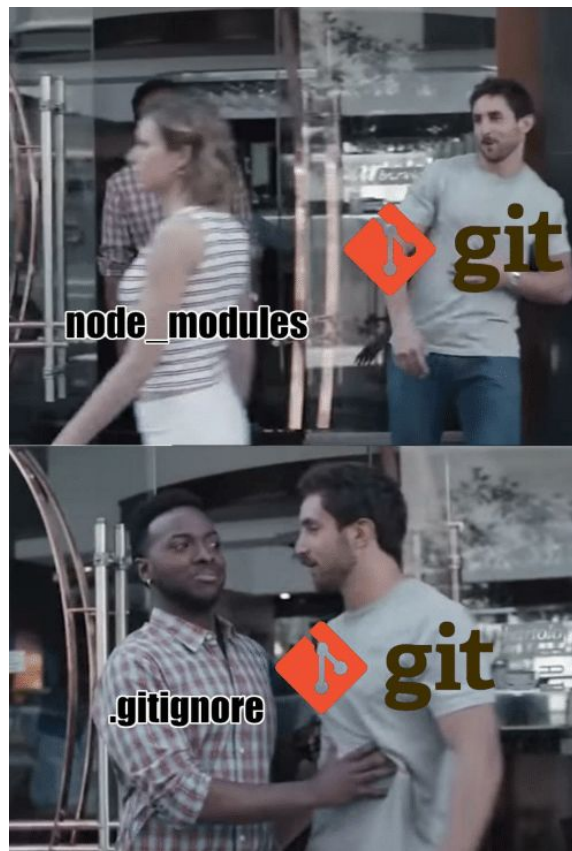
- Una vez instalado el paquete, ya se puede usar en el código del proyecto con "require()"

```
var lodash = require('lodash');  
  
var output = lodash.without([1, 2, 3], 1);  
console.log(output);
```

Node Package Manager (NPM)

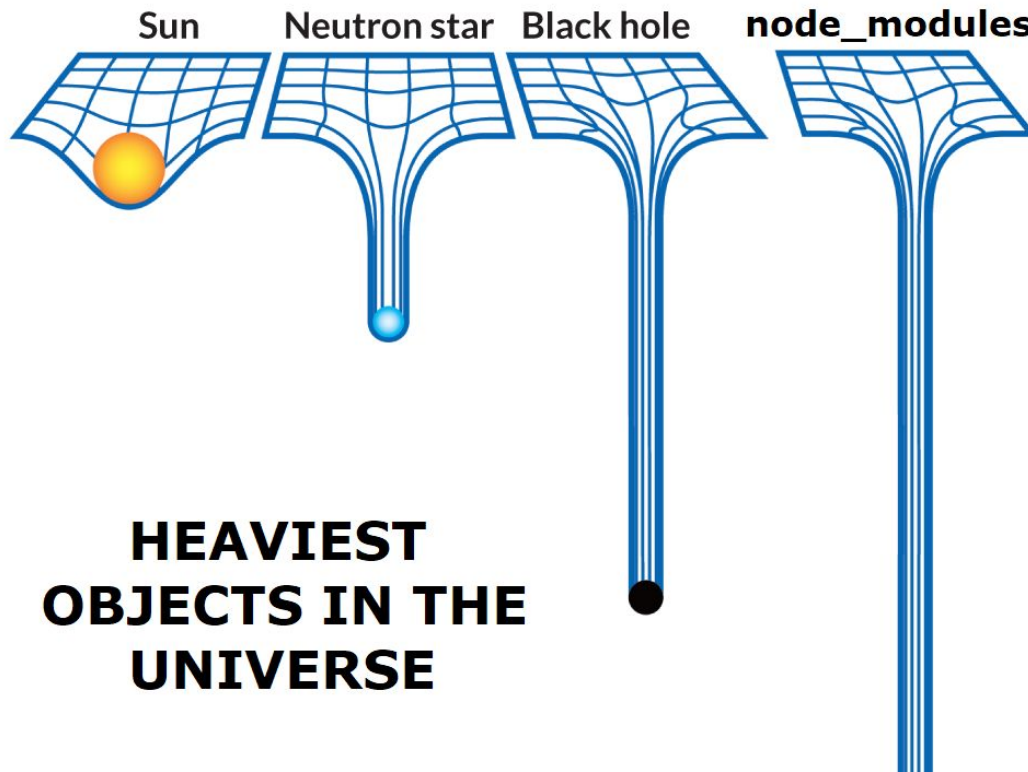
- Los paquetes descargados se guardan en la carpeta **node_modules**
- Esta carpeta se suele incluir en el **.gitignore** porque estos paquetes no se suben al repositorio (porque se pueden descargar en cualquier momento desde la red con **npm install**)

Node Package Manager (NPM)



You have over 5000 changes in
node_modules

Node Package Manager (NPM)



Node Package Manager (NPM)

- Se puede modificar el package.json y descargar automáticamente el paquete con

```
$ npm install --save left-pad
```

- Una ventaja es que obtiene automáticamente la última versión de la librería disponible en NPM

Node Package Manager (NPM)

- Se pueden instalar paquetes que contienen **herramientas para desarrollo**, no para ejecutar la aplicación: gulp, webpack, browserify, angular-cli...
- Estos paquete se especifican en la sección **"devDependencies"** del package.json

```
{
  "name": "my-awesome-package",
  "version": "1.0.0"
  "dependencies": {
    "lodash": "4.17.15"
  }
  "devDependencies": {
    "typescript": "3.7.2"
  }
}
```

Node Package Manager (NPM)

- Es habitual que algunas herramientas se instalen globalmente en el sistema, no de forma concreta en un proyecto (en linux necesitan **sudo**)

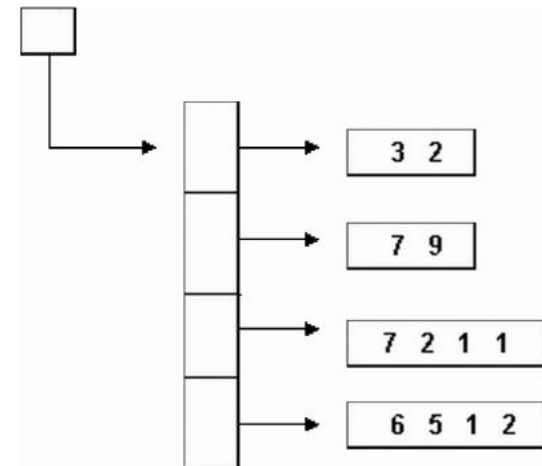
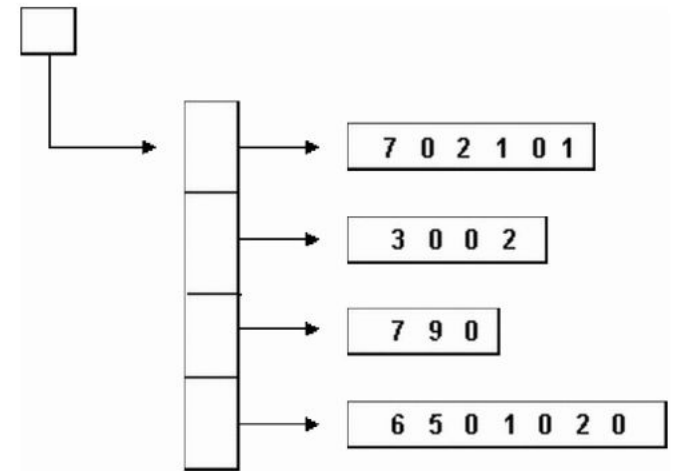
```
$ npm install -g @angular/cli
```

- Estas herramientas se registran de forma automática en el path y se pueden ejecutar en cualquier carpeta

```
$ ng --version
```

Ejercicio 3

- Reimplementa el ejercicio 2 para usar lodash para eliminar los ceros de los arrays



Módulos EcmaScript (ES Módulos)

- El soporte en Node se ha diseñado para que la **convivencia y transición** entre módulos **CommonJS** y **ES** sea lo más suave posible
- El objetivo es que una misma **aplicación** pueda tener **módulos de ambos tipos**
- Esta transición obliga a que existan **muchas situaciones** y **muchos detalles** a tener en cuenta (complejo)

Módulos EcmaScript (ES Modules)

- Sólo estudiaremos el tipo más sencillo de aplicación:
 - Todos sus módulos ES
 - Usa módulos de la API de Node como módulos ES
 - Usa módulos de librerías externas implementados en CommonsJS

Módulos EcmaScript (ES Modules)

ejem3

- Existen dos formas de implementar módulos ES
- 1) Extensión .mjs
- 2) Tipo "module" en el package.json

package.json

```
{  
  "name": "app",  
  "version": "1.0.0",  
  "type": "module"  
}
```


Módulos EcmaScript (ES Modules)

ejem3

- Conversión de módulos CommonsJS a ES
- 1) require() -> import

hello.js

```
console.log('Hello world');
```

app.js

```
require('./hello.js');
```



app.js

```
import './hello.js';
```

La extensión del fichero **NO** se puede omitir

Módulos EcmaScript (ES Modules)

ejem3

- 2) Para exportar **una única función anónima (default export)**. Se le da nombre al importarla

bar.js

```
module.exports = function() {  
  console.log('bar!');  
}
```



bar.js

```
export default function() {  
  console.log('bar!');  
}
```

app.js

```
var bar = require('./bar.js');  
bar();
```



app.js

```
import bar from './bar.js';  
bar();
```

Módulos EcmaScript (ES Modules)

ejem3

- 3) Para exportar una función con nombre

fiz.js

```
exports.fiz = function(){  
  console.log('fiz!');  
}
```



fiz.js

```
export function fiz(){  
  console.log('fiz!');  
}
```

app.js

```
var module = require('./fiz.js');  
module.fiz();
```



app.js

```
import * as module from './fiz.js';  
module.fiz();
```

Módulos EcmaScript (ES Modules)

ejem3

- 4) Otra forma de importar una **función con nombre**

fiz.js

```
exports.fiz = function() {  
  console.log('fiz!');  
}
```



fiz.js

```
export function fiz() {  
  console.log('fiz!');  
}
```

app.js

```
var fiz = require('./fiz.js').fiz;  
fiz();
```



app.js

```
import { fiz } from './fiz.js';  
fiz();
```

Módulos EcmaScript (ES Módulos)

ejem3

- 5) Se pueden exportar varios elementos y de diferentes tipos (funciones, objetos, valores, clases...)

utils.js

```
var obj = {  
  prop1: 3,  
  prop2: 'value'  
}  
  
function log(msg) {  
  console.log(msg);  
}  
  
class Date {  
  constructor(date) {  
    this.date = date;  
  }  
}  
  
exports.obj = obj;  
exports.log = log;  
exports.Date = Date;
```



utils.js

```
export var obj = {  
  prop1: 3,  
  prop2: 'value'  
}  
  
export function log(msg) {  
  console.log(msg);  
}  
  
export class Date {  
  constructor(date) {  
    this.date = date;  
  }  
}
```

Módulos EcmaScript (ES Modules)

ejem3

- 5) Se pueden exportar varios elementos y de diferentes tipos (funciones, objetos, valores, clases...)

app.js

```
var utils = require('./utils.js');  
  
utils.log('message');  
  
console.log(utils.obj);  
  
var date = new utils.Date('01-12-19');
```



app.js

```
import * as utils from './utils.js';  
  
utils.log('message');  
  
console.log(utils.obj);  
  
var date = new utils.Date('01-12-19');
```

Módulos EcmaScript (ES Modules)

ejem3

- 5) Se pueden exportar varios elementos y de diferentes tipos (funciones, objetos, valores, clases...)

app.js

```
var utils = require('./utils.js');  
  
utils.log('message');  
  
console.log(utils.obj);  
  
var date = new utils.Date('01-12-19');
```



app2.js

```
import { log, obj, Date } from './utils.js';  
  
log('message');  
  
console.log(obj);  
  
var date = new Date('01-12-19');
```

Módulos EcmaScript (ES Modules)

ejem3

- 6) Además de los elementos con nombre, también se puede exportar un objeto “default”

utils.js

```
export var obj = {  
  prop1: 3,  
  prop2: 'value'  
}  
  
export function log(msg) {  
  console.log(msg);  
}  
  
export class Date {  
  constructor(date) {  
    this.date = date;  
  }  
}  
  
export default { obj, log, Date }
```

app.js

```
import * as utils from './utils.js';  
  
utils.log('message');  
console.log(utils.obj);  
var date = new utils.Date('01-12-19');
```

app2.js

```
import utils from './utils.js';  
  
utils.log('message');  
console.log(utils.obj);  
var date = new utils.Date('01-12-19');
```


Módulos EcmaScript (ES Módules)

ejem4

- Las librerías de Node también se pueden importar como módulos ES

app.js

```
var lodash = require('lodash');  
  
var output = lodash.without([1, 2, 3], 1);  
console.log(output);
```



app.js

```
import lodash from 'lodash';  
  
var output = lodash.without([1, 2, 3], 1);  
console.log(output);
```

Módulos EcmaScript (ES Modules)

ejem4

- Un export default siempre se puede importar como *** as name**

app.js

```
var lodash = require('lodash');  
  
var output = lodash.without([1, 2, 3], 1);  
console.log(output);
```



app.js

```
import lodash from 'lodash';  
  
var output = lodash.without([1, 2, 3], 1);  
console.log(output);
```

Módulos EcmaScript (ES Modules)

ejem4

- No siempre se puede acceder a los elementos individuales con un **import**

app.js

```
var lodash = require('lodash');  
  
var output = lodash.without([1, 2, 3], 1);  
console.log(output);
```



app.js

```
import { without } from 'lodash';  
  
var output = without([1, 2, 3], 1);  
console.log(output);
```

ERROR

lodash no exporta un elemento
“without”. Exporta un objeto “default”

Ejercicio 4

- Cambia el ejercicio 3 para que use módulos ES en vez de CommonJS

