keep coding

Fullstack Web Bootcamp

# Desarrollo Backend con Node.js

20/10/2025

# Índice

keep coding

# Introducción

Desarrollador Fullstack, exalumno (reincidente) de KeepCoding, amante de Typescript y todo el ecosistema Javascript.
Entusiasta de la formación y divulgación en todo aquello que esté relacionado con la tecnología y el emprendimiento.

# Requisitos

- Conocer VSCode
- Conocer GIT
- Comprender los conceptos del módulo de desarrollo frontend con Javascript
- Ganas de aprender

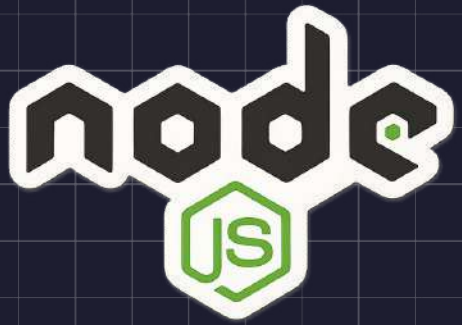
keep coding

# Objetivos

- Entender cómo funciona la arquitectura básica de un entorno web
- Conocer cómo se comunica nuestro navegador con el entorno
- Entender las peticiones HTTP y sus verbos
- Comprender cómo se persisten los datos
- Aplicar una política correcta de permisos
- Disfrutar por el camino

keep coding

# Stack y Herramientas


keep coding

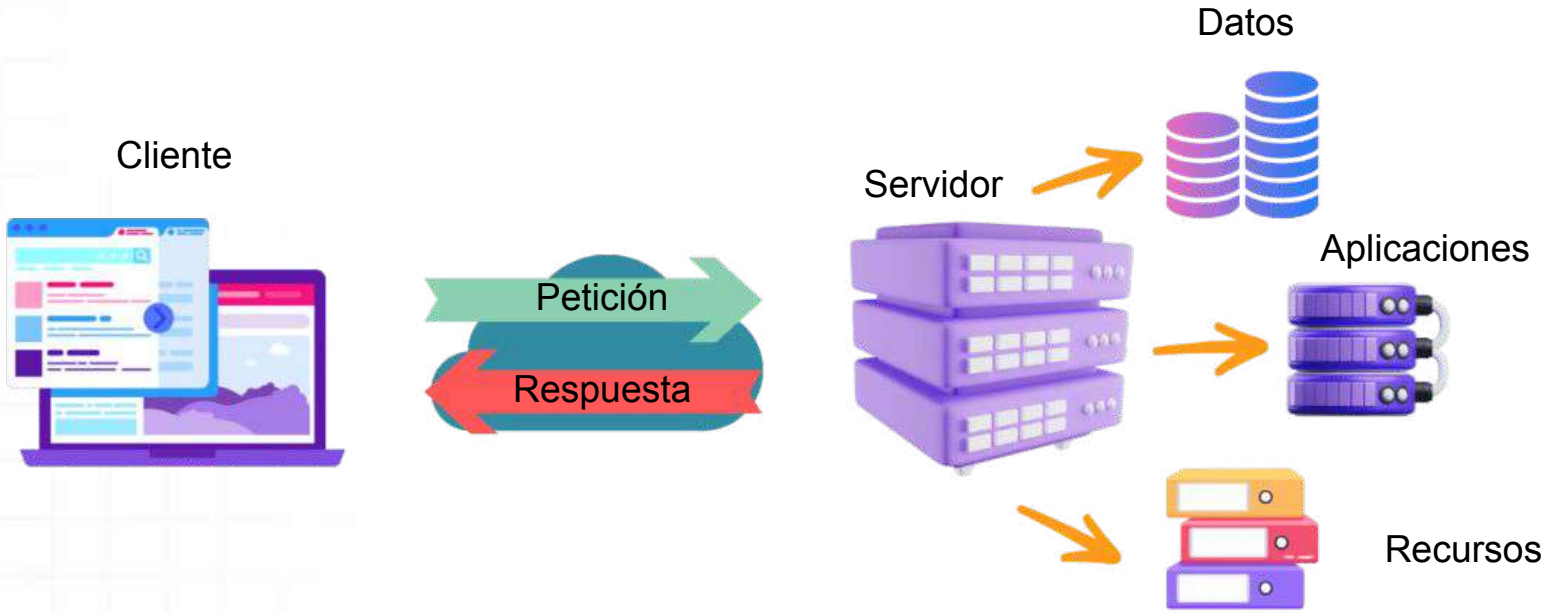
# ¿Qué construiremos?

Api RestFull con autenticación

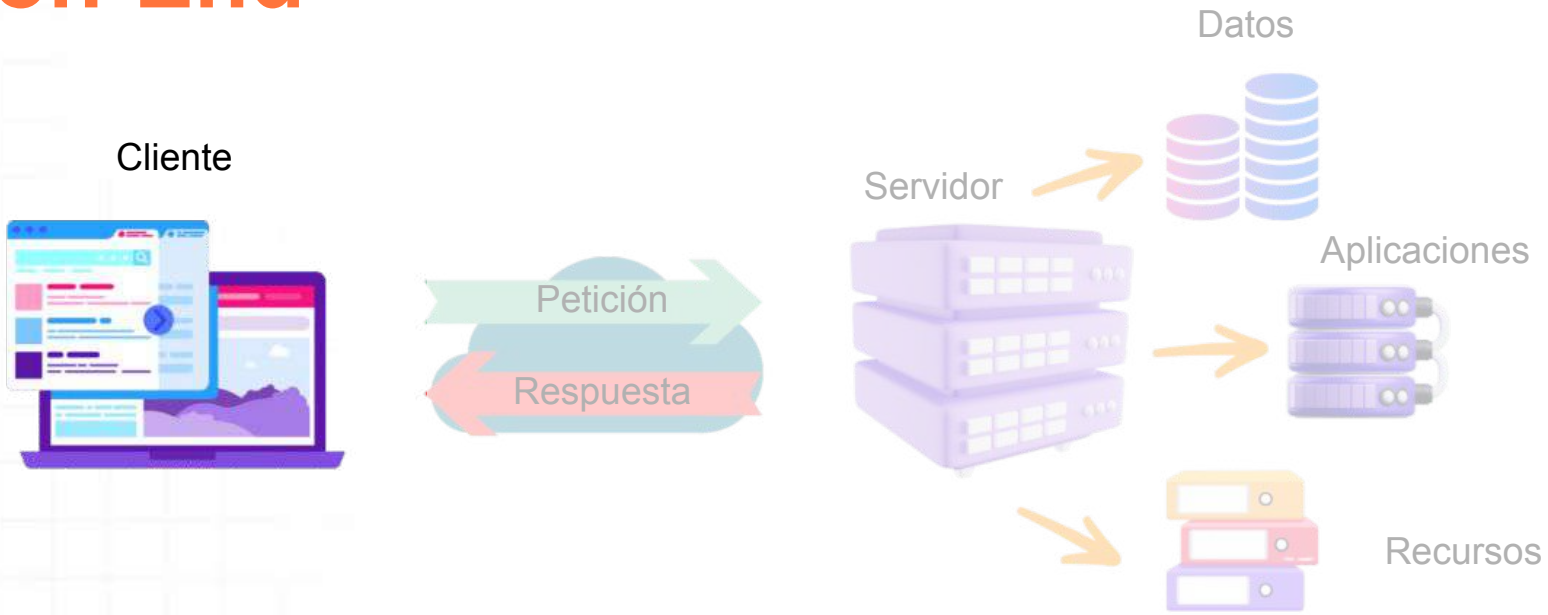
keep coding

# Web

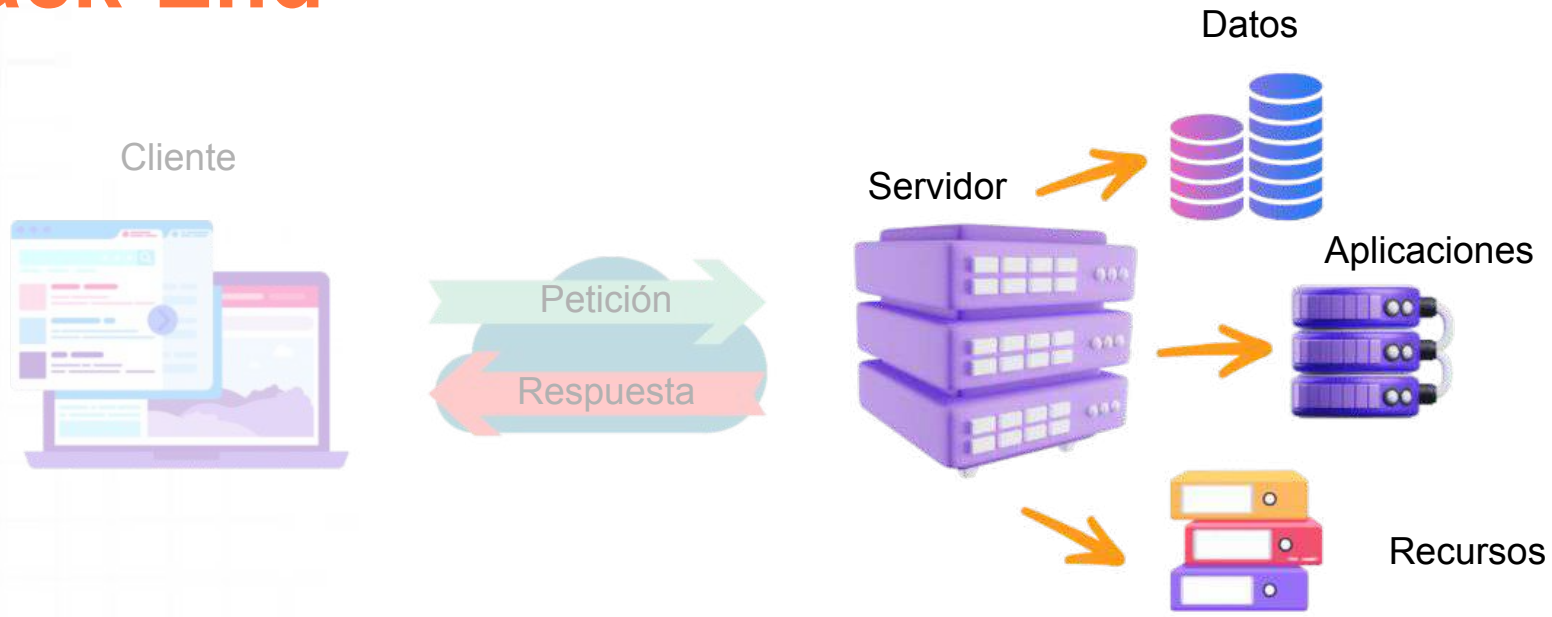
# Cómo funciona la web



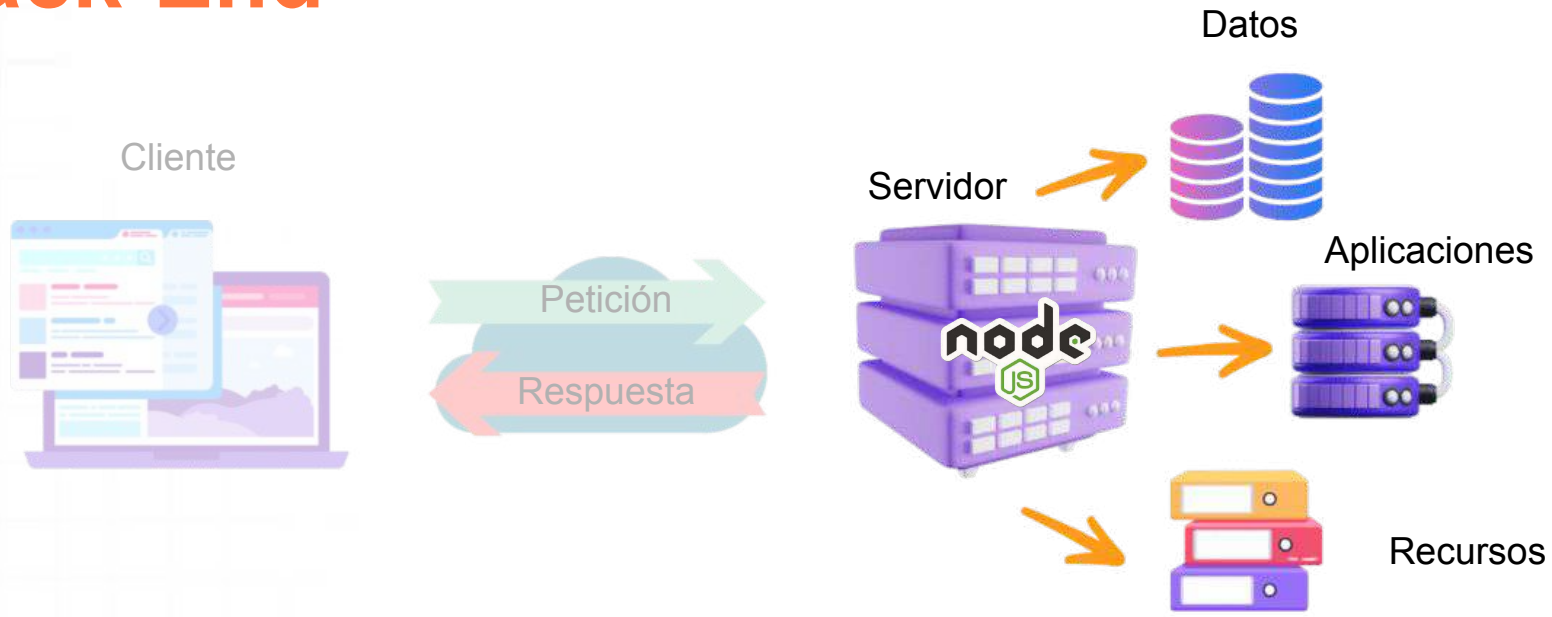
# Cómo funciona la web
## Fron-End



# Cómo funciona la web
## Back-End



# Cómo funciona la web
## Back-End



keep coding

# Intro NodeJS

# NodeJS
## ¿Qué es?

- Es un intérprete de Javascript
- Inicialmente diseñado para servidor
- Orientado a eventos*
- Con servidor de aplicación*
- Basado en el motor V8*

# NodeJS
## Orientado a eventos

- En la programación secuencial, es el programador quien decide el flujo y el orden de ejecución.
- En la programación orientada a eventos, el usuario o los programas cliente son quienes deciden el flujo.

# NodeJS
## Servidor de aplicación

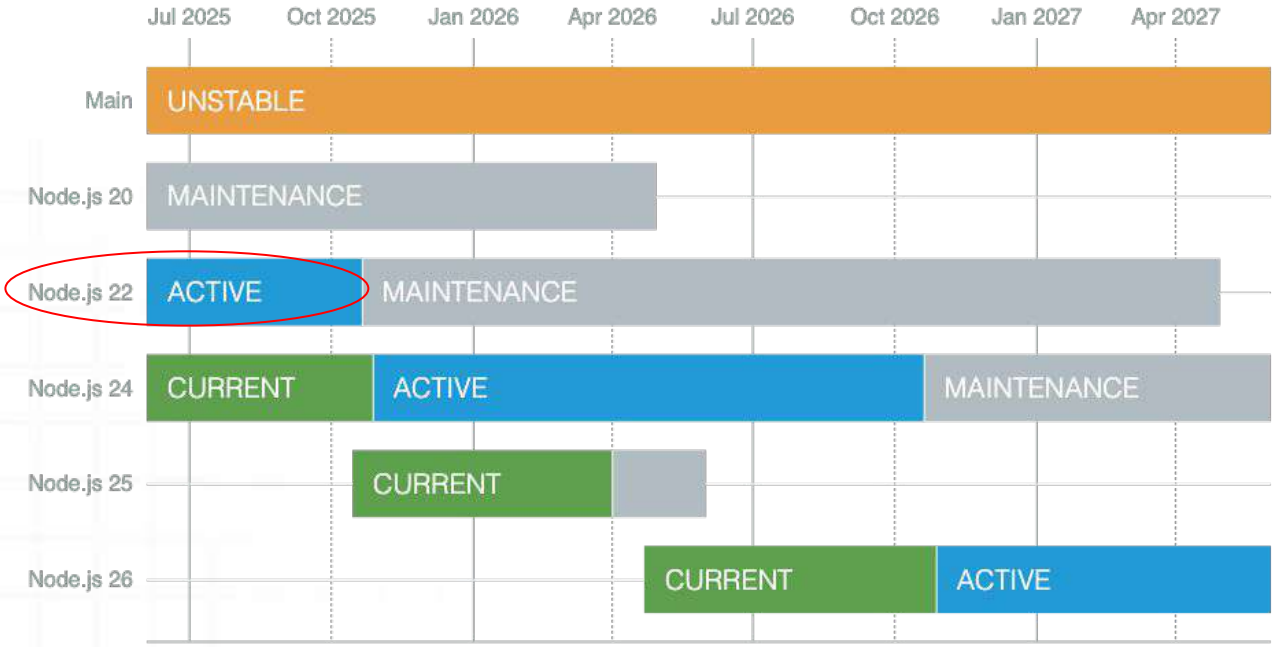
- No necesitamos un programa que ejecute nuestro código como Tomcat, IIS, etc.
- Tampoco necesitamos un servidor web como Apache, nginx, etc.
- Nuestra aplicación realiza todas las funciones de un servidor.

# NodeJS
## Motor V8

- Motor de Javascript creado por Google para Chrome
- Escrito en C++
- Multiplataforma (Windows, Linux, Mac)

keep coding

# NodeJS
# Versiones



# NodeJS
## Ventajas

Node.js tiene grandes ventajas reales en:

- Aplicaciones de red, como APIs , servicios en tiempo real, servidores de comunicaciones, etc.
- Aplicaciones cuyos clientes están escritos en Javascript, pues compartimos código y estructuras entre el servidor y el cliente.

https://node.green/

# NodeJS
# Instalación

Existen distintas formas de instalar Node.js

- Desde el instalador oficial
  - Más sencillo de instalar
- Con un instalador de paquetes
  - Más fácil desinstalar y mantener actualizado
- Con un gestor de versiones
  - Podemos gestionar distintas versiones

# NodeJS
## Versions Managers

Un gestor de versiones como NVM nos permite tener distintas instalaciones aisladas de Node.js en nuestro dispositivo e utilizarlas de manera independiente según el proyecto.

Windows: https://github.com/coreybutler/nvm-windows
Linux / Mac: https://github.com/nvm-sh/nvm

# Instalar node.js

```
$ nvm --version

$ nvm list

$ nvm install <version>

$ nvm use <version>

$ node --version
```

keep coding

# NodeJS
## NVM - Cambio automático

https://github.com/nvm-sh/nvm#nvmrc

```
$ node --version > .nvmrc
```

Posteriormente, al entrar en la carpeta podemos ejecutar

```
$ nvm use
```

Y tras salir de la carpeta

```
$ nvm use default
```

Para usuarios de Mac y Linux:
https://github.com/nvm-sh/nvm#deeper-shell-integration

keep coding

# NodeJS
## NVM - Alternativas

- n es una alternativa de nvm de larga data que logra lo mismo con comandos ligeramente diferentes y se instala a través de npm en lugar de un script bash.
- fnm es un administrador de versiones más reciente, que afirma ser mucho más rápido que nvm. (También usa Azure Pipelines).
- Volta es un nuevo administrador de versiones del equipo de LinkedIn que afirma una velocidad mejorada y soporte multiplataforma.
- asdf-vm es una única CLI para varios idiomas, como gvm, nvm, rbenv y pyenv (y más), todo en uno.
- nvs (Node Version Switcher) es una alternativa a nvm multiplataforma con la capacidad de integrarse con VS Code.

https://pavel-romanov.com/5-node-version-managers-compared-which-is-right-for-you

keep coding

# Node.js

## Servidor Básico

# Talk is cheap

Show me the code

# Crear un servidor básico

```
$ node index.js

# Y si actualizamos?

$ npm i -g nodemon
$ npx nodemon
```

# Node.js

## Gestor de paquetes NPM

# NodeJS
# NPM

**Node Package Manager** es un gestor de paquetes que nos ayuda a gestionar las dependencias de nuestro proyecto.

Entre otras cosas nos permite:

- Instalar librerías o programas de terceros
- Eliminarlas
- Mantenerlas actualizadas

Generalmente se instala conjuntamente con Node.js de forma automática.

keep coding

# NodeJS
## NPM - package.json

Se apoya en un fichero llamado package.json para guardar el estado de las librerías.

```
$ npm init
```

Crea este fichero.

Documentación en https://docs.npmjs.com/cli/v11/configuring-npm/package-json

# package.json

```
{
  "name": "myapp",
  "version": "1.0.0",
  "description": "Esta es la descripción",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Nauel Gómez",
  "license": "ISC"
}
```

keep coding

# NodeJS
## NPM - global vs local

Instalación local, en la carpeta del proyecto

```
$ npm install <paquete>
```

Instalación global, en nuestro sistema, o en la carpeta de instalación de node.

```
$ npm install -g <paquete>
```

Si el paquete tiene ejecutables, se hará un vínculo a ellos en /usr/local/bin

keep coding

# NodeJS
# NPM - npx

Npx nos permite utilizar paquetes locales o **remotos**, en caso de ser remotos se instalan temporalmente.

```
$ npx paquete
```

Si está en local se utiliza, si no se descarga (~/.npm/_npx), por ejemplo:

```
$ npx nodemon
```

# Talk is cheap

Show me the code

# Instalar un módulo

Instalar el módulo chance y usar su generador de nombres en el servidor básico.
https://www.npmjs.com/package/chance?activeTab=readme

```
[npm init]

npm install chance
```

keep coding

# JS Pro

keep coding

# JS Pro

## Hoisting

keep coding

# JS Pro
# Hoisting

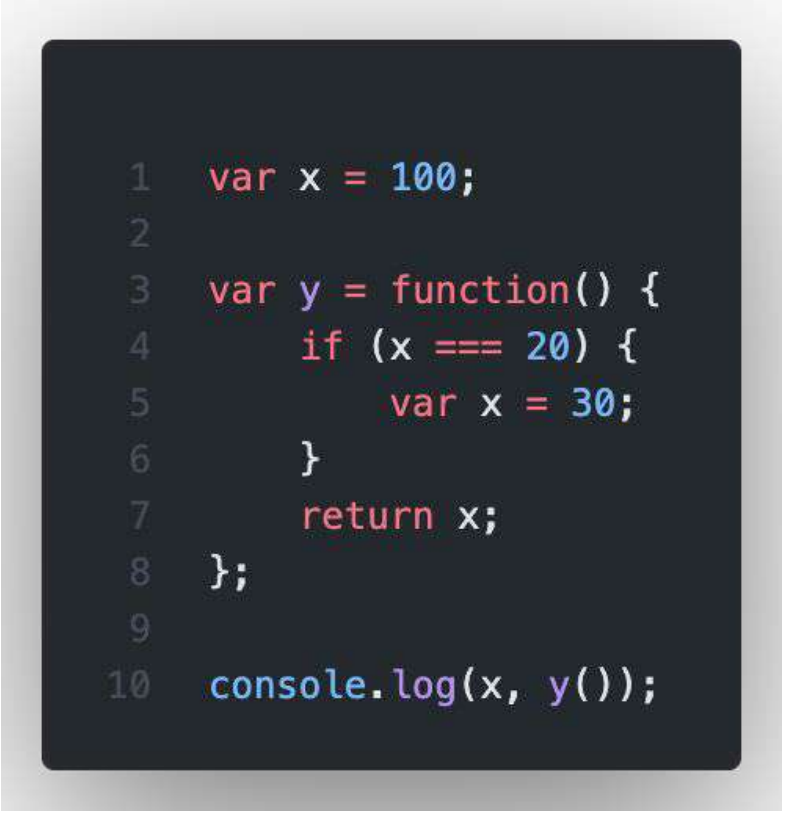
Las declaraciones de variables en JS son “hoisted”. Esto significa que el intérprete va a mover al principio de su contexto (función) la declaración, manteniendo la inicialización donde estaba.

```
var pinto = "My Value";

function pinta() {
    // var pinto; // Declaración hoisted
    console.log("Pinto: " + pinto); // My Value
    var pinto = "Local Value"; // Esto hará hoisting de pinto y será undefined
}

pinta();
```

keep coding

# JS Pro
# Hoisting

Qué valores se escribirán en la consola?



```
1   var x = 100;
2
3   var y = function() {
4       if (x === 20) {
5           var x = 30;
6       }
7       return x;
8   };
9
10  console.log(x, y());
```

keep coding

# JS Pro
# Hoisting

Qué valores se escribirán en la consola?

```
1   var x = 100;
2
3   var y = function() {
4       var x; // -> Hoisting, ahora es undefined
5       if (x === 20) {
6           // Aquí se mantiene la inicialización
7           x = 30;
8       }
9       return x; // X es undefined
10  };
11
12  console.log(x, y());
13  // Output: 100 undefined
```

keep coding

# JS Pro

## JSON

# JS Pro
# JSON

Javascript Object Notation

- Es un formato para intercambio de datos, derivado de la notación literal de objetos en Javascript.
- Se utiliza habitualmente para serializar objetos o estructuras de datos.
- Se ha popularizado mucho principalmente como alternativa a XML, por ser más ligero que éste.

keep coding

# JS Pro
# JSON

Convirtiendo un objeto a JSON

```
var empleado = {
    nombre: "Thomas Anderson",
    profesion: "Developer",
};

JSON.stringify(empleado);
// Output: '{"nombre":"Thomas Anderson","profesion":"Developer"}'
```

keep coding

# JS Pro
# JSON

Convirtiendo JSON a un objeto

```
var textoJSON = '{"nombre":"Thomas Anderson","profesion":"Developer"}';

var objeto = JSON.parse(textoJSON);
// Output: Object { nombre: 'Thomas Anderson', profesion: 'Developer' }
```

keep coding

# JS Pro

‘use strict’

# JS Pro
# Modo estricto

El modo Strict habilita más avisos y hace Javascript un lenguaje un poco más coherente. El modo no estricto se suele llamar “sloppy mode”. Para habilitarlo se puede escribir al principio de un fichero:

```
‘use strict’;
```

También se puede habilitar solo para una función:

```
function estoyEnStrictMode() {
    ‘use strict’;
    ...
}
```

# JS Pro
# Modo estricto

Algunos ejemplos de los beneficios del modo estricto:

- **Las variables deben ser declaradas. En *sloppy mode*, una variable mal escrita se crearía global, en *strict* falla.**
- Reglas menos permisivas para los parámetros de funciones, por ejemplo no se pueden repetir.
- Los objetos de argumentos tienen menos propiedades (arguments.callee por ejemplo)
- **En funciones que no son métodos, this será undefined.**
- Asignar y borrar propiedades inmutables fallará con una excepción, en *sloppy mode* fallaba silenciosamente.
- No se pueden borrar identificadores si cualificar (delete variable; —> delete this.variable;)
- **eval() es más limpio. Las variables que se definen en el código evaluado no pasan al scope que lo rodea.**

keep coding

# JS Pro

## Funciones

keep coding

# JS Pro
# Funciones

Las funciones son objetos

Pero también tienen propiedades y métodos

keep coding

# JS Pro
# Funciones - Declaración

- Requieren un nombre
- Solo a nivel de programa o directamente en el cuerpo de otra función.
- Hacen Hoisting

```
1  function suma(numero1, numero2) {
2      return numero1 + numero2;
3  };
```

keep coding

# JS Pro
# Funciones - Expresión

- Cómo son expresiones, se pueden definir en cualquier sitio donde pueda ir un valor.
- No hacen hoisting, se pueden utilizar sólo después de su definición.
- Pueden tener un nombre, pero solo sería visible dentro de su cuerpo.

```
1  var suma = function(numero1, numero2) {
2      return numero1 + numero2;
3  };
```

keep coding

# JS Pro
# Funciones - Métodos

- Cuándo una función es una propiedad de un objeto se llama **método.**

```
var calculadora = {
    suma: function(numero1, numero2) {
        return numero1 + numero2;
    },
    resta: function(numero1, numero2) {
        return numero1 - numero2;
    }
};
```

keep coding

# JS Pro
# Funciones - Instancias

Cuándo se usa **new** al invocar una función se comporta como un constructor de objetos.

```
function Fruta() {
    var nombre, familia;
    this.getNombre = function() {
        return nombre;
    };
    this.setNombre = function(valor) {
        nombre = valor;
    };
};
```

keep coding

# JS Pro

## Callbacks

# JS Pro
# Callbacks

Un ejemplo es cuando usamos setTimeout, que recibe como parámetros:

- Una función con el código que queremos que ejecute tras la espera.
- El número de milisegundos que tiene que esperar para llamarla.

```
1  console.log('Empiezo');
2  setTimeout(function() {
3      console.log('He terminado');
4  }, 3000);
```

keep coding

# JS Pro
# Callbacks

En Node.js todos los usos de I/O (entrada / salida) deberían ser asíncronos.

Si tras una llamada a una función asíncrona queremos hacer algo, como comprobar su resultado o si hubo errores, le pasaremos **un argumento más**, una expresión de tipo función (callback), para que la invoque cuando termine.

# Talk is cheap

Show me the code

# Ejercicio

Hacer una función que reciba un texto y tras dos segundos lo escriba en la consola.

La llamaremos `escribeTras2Segundos`

keep coding

# Ejercicio

Llamarla dos veces (texto1 y texto2. Deben salir los textos con sus pausas correspondientes.

Al final escribir en la consola “Fin”.

Llamada 1 - 2sec. - **texto1** - llamada 2 - 2secs. - **texto2** - **Fin**

keep coding

# Ejercicio TODO

Repitamos la llamada a `escribeTras2Segundos` varias veces ejecutar un total de N veces?

keep coding

# Gracias

ngomez@codiara.com
https://www.linkedin.com/in/nauelg/




www.keepcoding.io cursos@keepcoding.io (+34) 916 33 1779

# Descanso

Volvemos 21:05