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

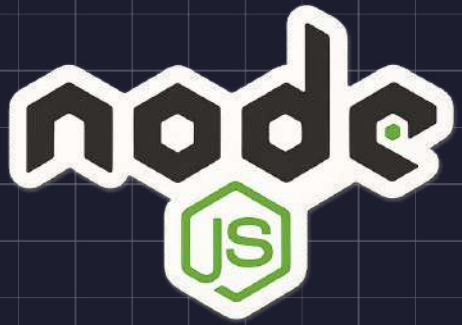
keep coding

# 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

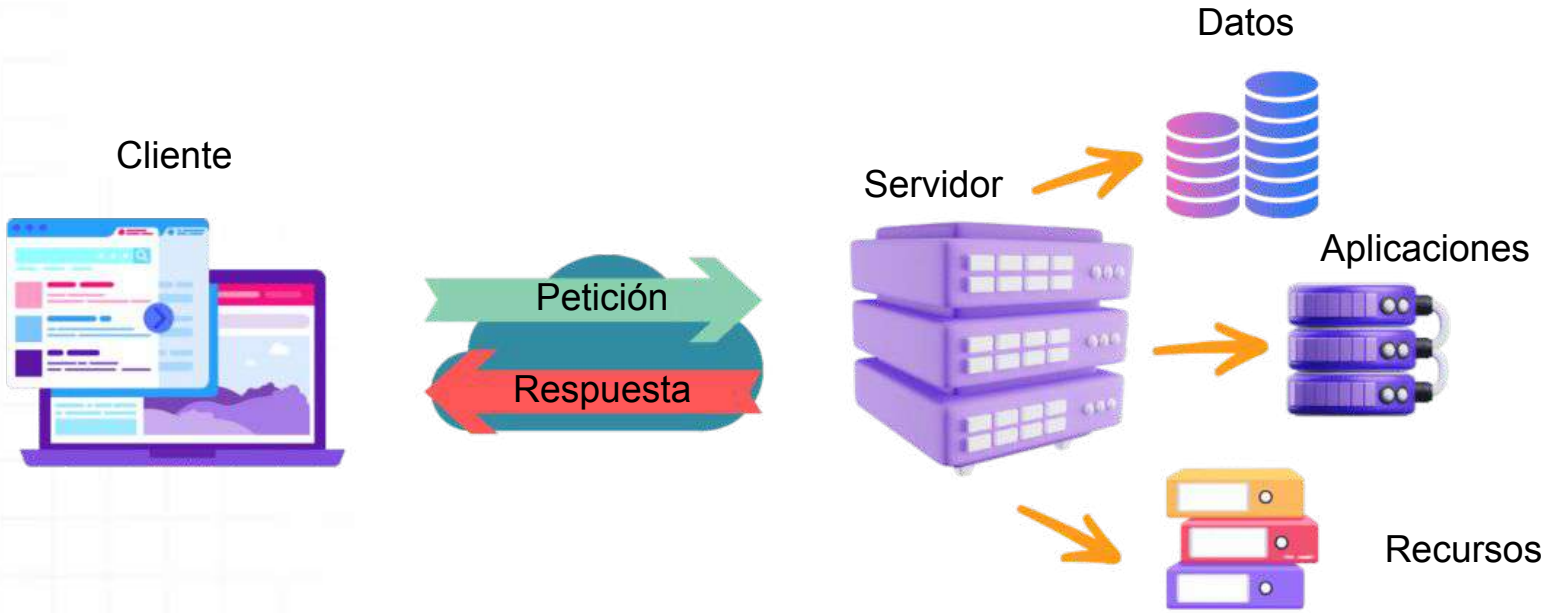
# ¿Qué construiremos?

Api RestFull con autenticación

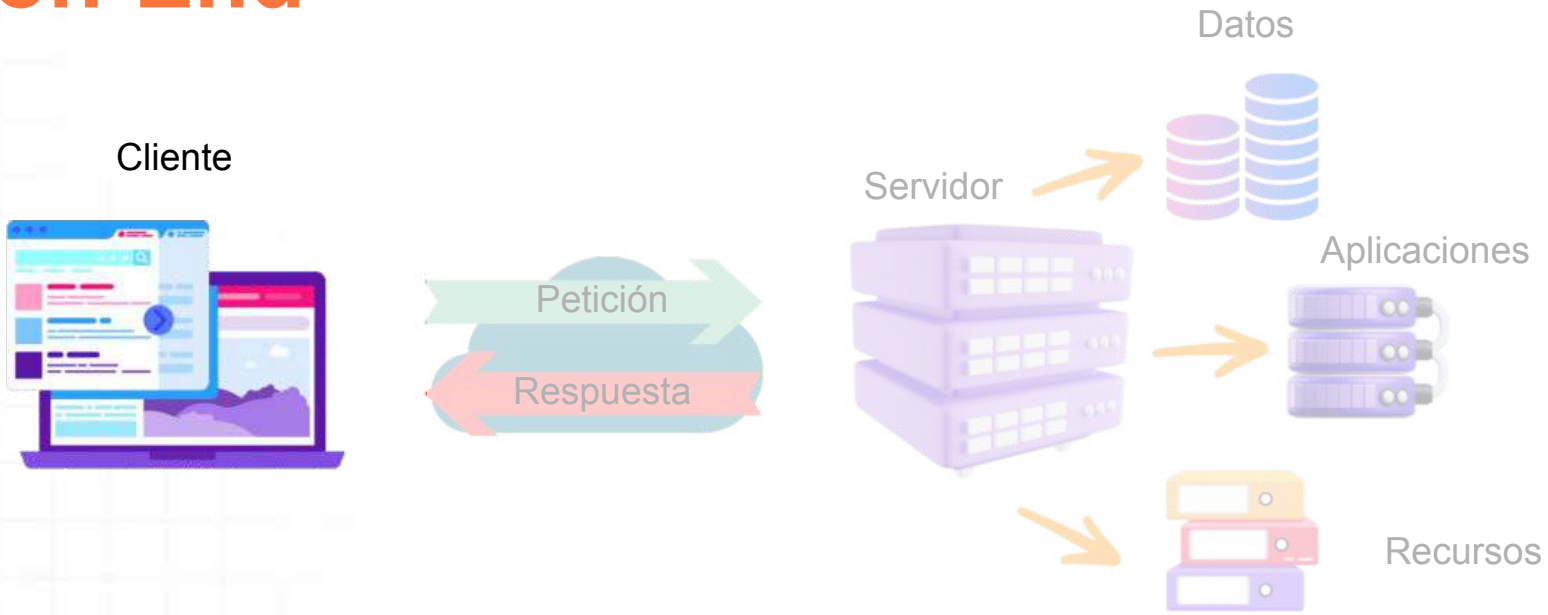
keep coding

# Web

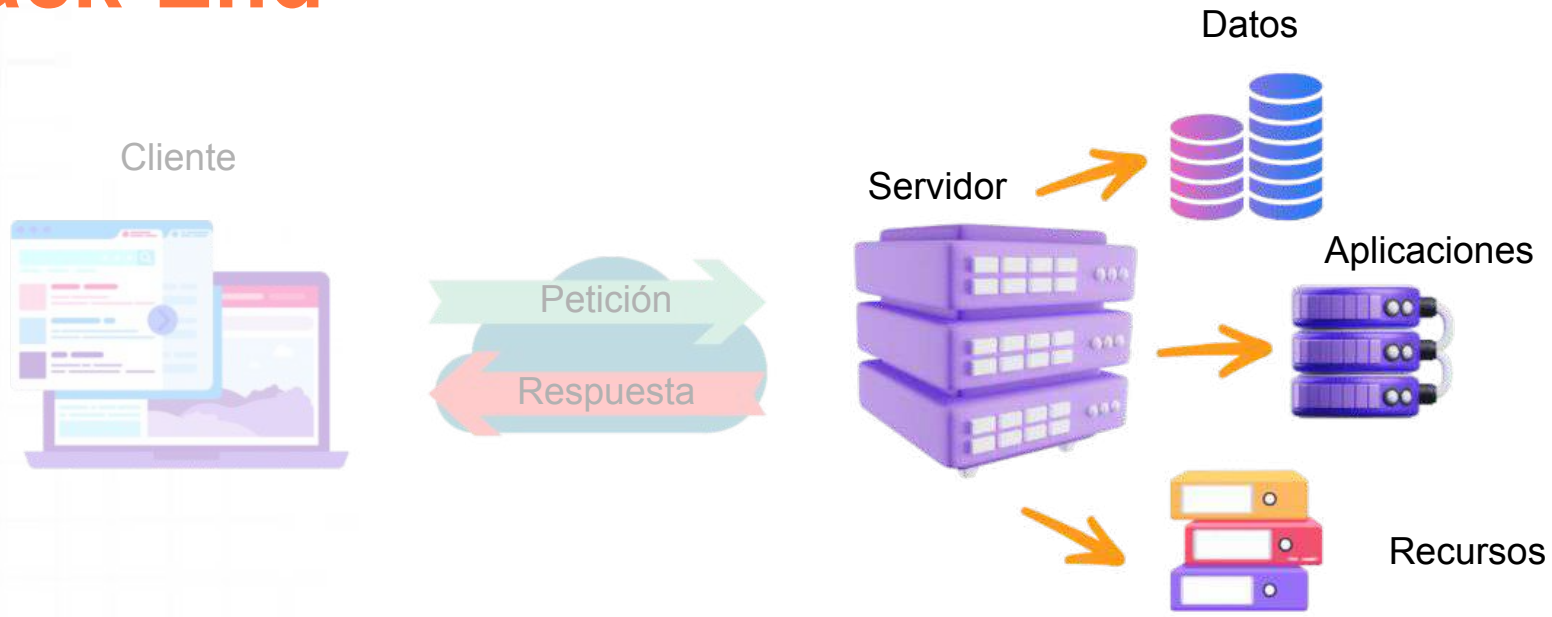
# Cómo funciona la web



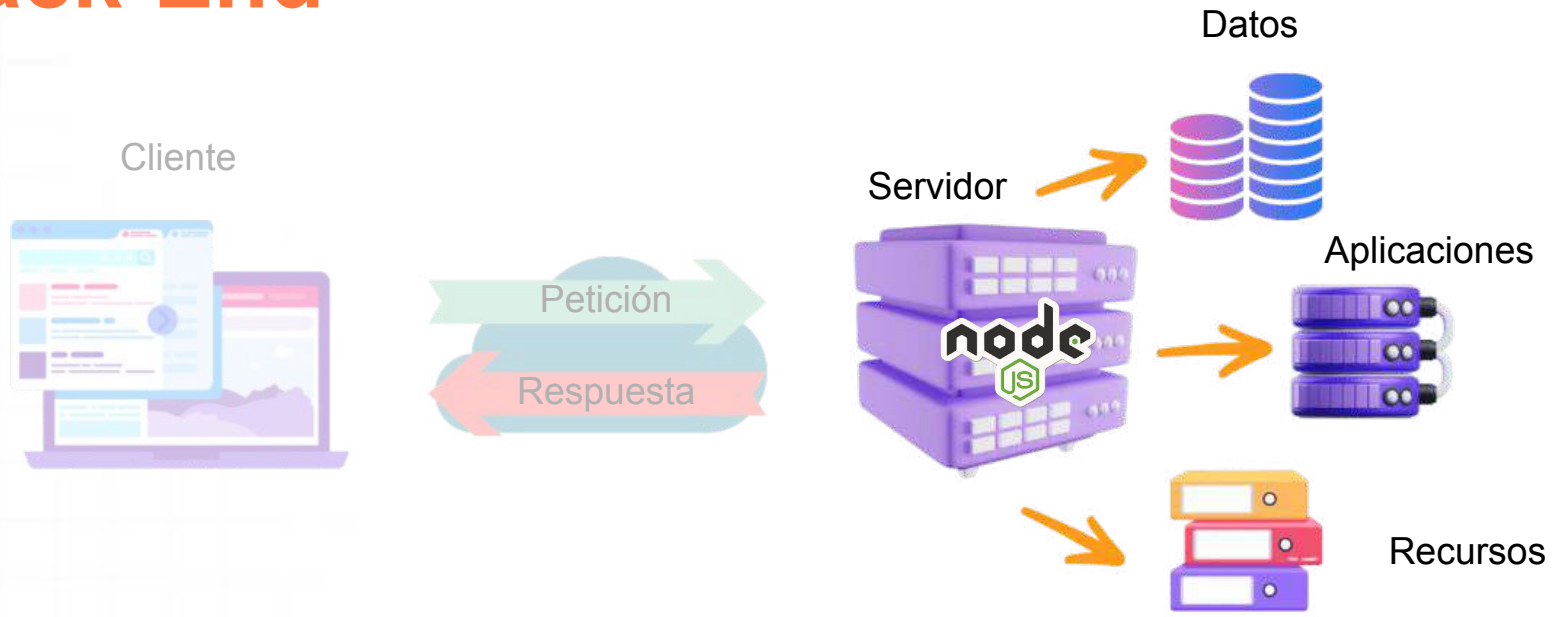
# Cómo funciona la web
## Fron-End



# 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

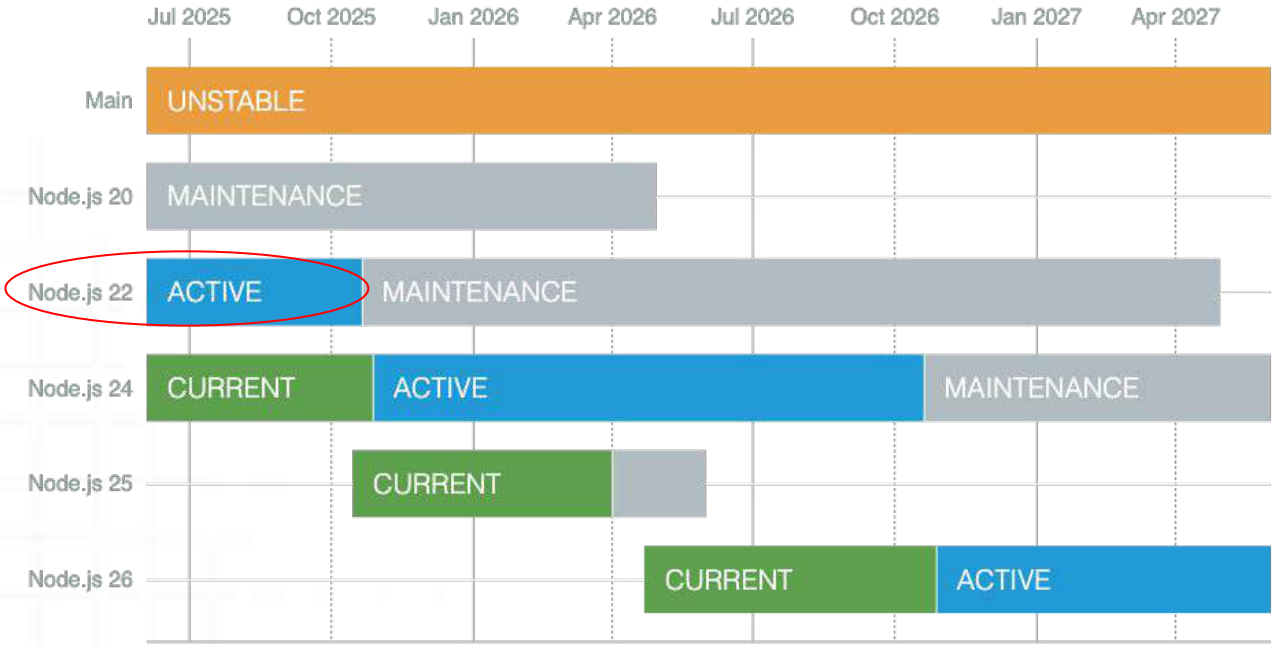
- No necesitamos un programa que ejecute nuestro código como Tomcat, IIS, etc.
- Tampoco necesitamos un servidor web como Apache, nginx, etc.
- Nuestra aplicación realiza todas las funciones de un servidor.

# 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

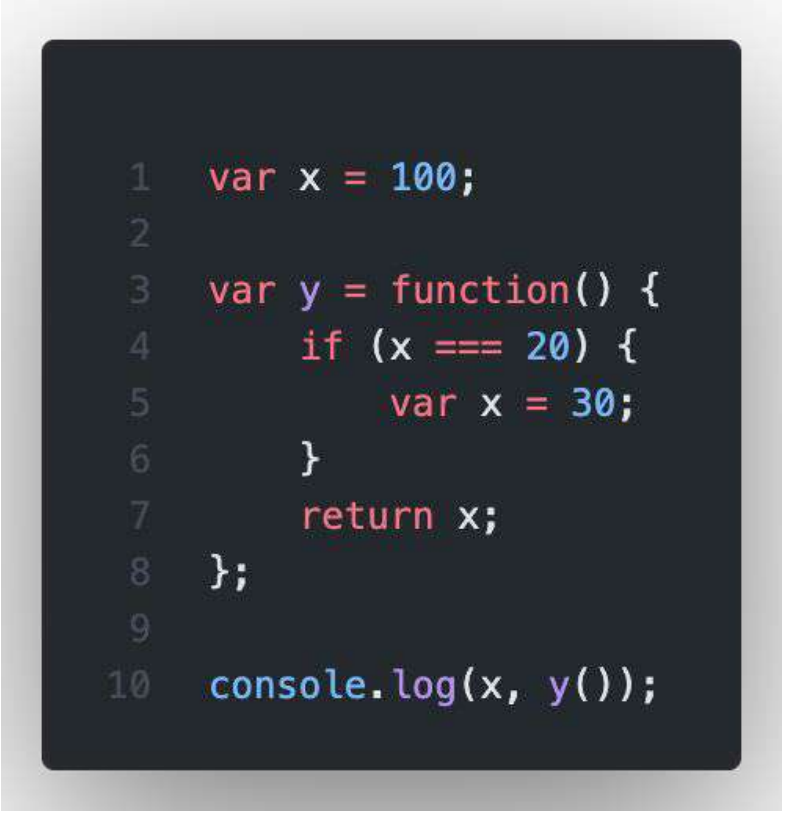
Las declaraciones de variables en JS son “hoisted”. Esto significa que el intérprete va a mover al principio de su contexto (función) la declaración, manteniendo la inicialización donde estaba.

```
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

# JS Pro

## Truthy and Falsy

keep coding

# JS Pro
# Truthy & Falsy

**En javascript todo tiene un valor booleano**, generalmente conocido como **truthy** o **falsy.**

keep coding

# JS Pro
## Truthy & Falsy

```
1  var variable = 'value';
2
3  if (variable) {
4      console.log('Soy truthy');
5  }
6
7  variable = 0;
8
9  if (variable) {
10     console.log('...');
11 } else {
12     console.log('Soy falsy');
13 }
```

keep coding

# JS Pro
# Truthy & Falsy

Los siguientes valores siempre son **falsy**:

- `false`
- 0 (cero)
- “” (cadena vacía).
- `null`
- `undefined`
- `NaN`

Todos los demás valores son **truthy**, incluyendo “0” (cero entre comillas) y “false” (false entre comillas), **empty functions**, **empty arrays** y **empty objects**.

keep coding

# JS Pro
## Truthy & Falsy

```
// Los valores false, 0 (cero), y "" (cadena vacía) son equivalentes:
var c = (false == 0); // true
var d = (false == ""); // true
var e = (0 == ""); // true

// Los valores null y undefined no son equivalentes con nada, excepto con ellos mismos:
var f = (null == false); // false
var g = (null == null); // true
var h = (undefined == undefined); // true
var i = (undefined == null); // true

// Y por último, el valor NaN no es equivalente con nada, ni siquiera consigo mismo:
var j = (NaN == null); // false
var k = (NaN == NaN); // false
```

keep coding

# JS Pro
## Truthy & Falsy - se complica

```
1  if ( [] ) { /*se ejecuta*/ }
2  /*Array es instancia de Object, y existe*/
3
4  if ( [] == true ) { /*no se ejecuta*/ }
5  /*comparamos valores!, [].toString -> "" -> falsy*/
6
7  if ( "0" == 0 ) // true (se convierten a números)
8
9  if ( "0" ) {console.log('si')} // "si" (se evalúa el string)
```

keep coding

# JS Pro
## Truthy & Falsy - solución?

Usamos el **igual estricto (===)** y el **distinto estricto (!==)**

```
1  var melio = ( false == 0 ); // true
2
3  var seguro = ( false === 0 ); // false
4  // Primero compara el tipo y luego el valor
```

keep coding

# JS Pro

## This

# JS Pro
# This

La palabra clave this trata de representar al **objeto que llama** a nuestra función cómo método, no donde está definida.

Por lo general, su valor hace referencia **al objeto propietario** de la función que la está invocando, o en su defecto, al objeto donde dicha función es un método.

keep coding

# JS Pro
# This - Cuidado

De forma general, cuando se usa en algo **distinto a un método** su valor es el contexto global (o undefined en modo estricto).

```
1  console.log(this); // window en un browser, global en node
2
3  function pinta() {
4      console.log(this); // window en un browser, global en node
5  }
```

keep coding

# JS Pro
# This

Como manejarlo? Le asignamos this con bind

```
1  var persona = {
2      name: 'Luis',
3      surnames: 'Gomez',
4      fullName: function() {
5          console.log(this.name + ' ' + + this.surnames);
6      }
7  }
8  setTimeout(persona.fullName.bind(persona), 1000);
```

keep coding

# JS Pro

## Clousures

# JS Pro
# Clousures

Un clousure se construye con una función (A) que devuelve otra (B).

La función devuelta (B), sigue manteniendo el acceso a todas las variables de la función que la creó (A).

keep coding

# JS Pro
## Clousures - Ejemplo

```
1  function creaClosure(valor) {
2      return function(){
3          return valor;
4      }
5  }
```

keep coding

# JS Pro

## Clases

keep coding

# JS Pro
# Clases

```
class Mascota {
    constructor(nombre) {
      this.nombre = nombre;
    }
    saluda() {
      console.log(`Hola soy ${this.nombre}`);
    }
}
```

keep coding

# JS Pro
# Clases

```
1  let perro = new Perro('Niebla');
2  perro.saluda();
```

keep coding

# JS Pro
## Clases

```
1  class Perro extends Mascota {
2      constructor(nombre) {
3          super(nombre);
4      }
5  }
6  let perro = new Perro('Niebla');
7  perro.saluda();
```

# Talk is cheap

Show me the code

keep coding

# JS Pro

## Process

keep coding

# JS Pro
# Process

El objeto global process tiene muchas propiedades que nos serán útiles, como process.platform, que en OS X nos dirá ‘darwin’, en linux ‘linux’, etc.

También tiene métodos útiles como process.exit(int) que para node estableciendo un exit code.

O eventos como process.on(‘exit’, callback) donde podemos hacer cosas antes de salir.

keep coding

# JS Pro

## Event Loop

# JS Pro
# Event Loop

Node usa un solo hilo.

Tiene un bucle interno, que podemos llamar ‘event loop’ donde cada vuelta ejecuta todo lo que tiene en esa ‘fase’, dejando los callbacks pendientes para otra vuelta.

La siguiente vuelta, mira a ver si ha terminado algún callback y si es así ejecuta sus handlers.

# JS Pro
# Event Loop - Non blocking

Si node se quedara esperando hasta que termine una query o una petición a Instagram, acumularía demasiados eventos pendientes y dejaría de antender a las siguientes peticiones, ya que **usa un solo hilo.**

Por eso todas las llamadas a funciones que usan IO, se hacen de forma asíncrona. Se aparcan para que nos avisen cuando terminen.

https://www.builder.io/blog/visual-guide-to-nodejs-event-loop

keep coding

# JS Pro
# Event Loop - Events

Node nos proporciona una forma de manejar IO en forma de eventos.

Usando el EventEmitter, podemos colgar eventos de un identificador.

eventEmitter.on('llamar telefono', suenaTelefono);

# JS Pro
# Event Loop - Events

Y podemos emitir el identificador cuando queremos que sus eventos ‘salten’.

eventEmitter.emit(‘llamar telefono’);

# JS Pro
# Event Loop - Events

Son cómodos para procesar streams.

Mandando a un fichero todo lo que se escriba en la consola.

process.stdin.on(‘data’, () => {...})

keep coding

# JS Pro

## Módulos

keep coding

# JS Pro
# Modulos

Los módulos de Node.js se basan en el estándar de **CommonJS**.

- Los módulos usan **exports** para exportar cosas.
- Quien quiere usar un módulo lo carga con **require**.
- La instrucción `require` es **síncrona.**
- Un módulo es un **singleton.**

keep coding

# JS Pro
# Modulos

donde busca Node.js los módulos:

1. Si es un módulo del core (node:...=
2. Si es local (la url es relativa).
3. Módulos de la carpeta node_modules local
4. Módulos de la carpeta node_modules global

# JS Pro
# Modulos

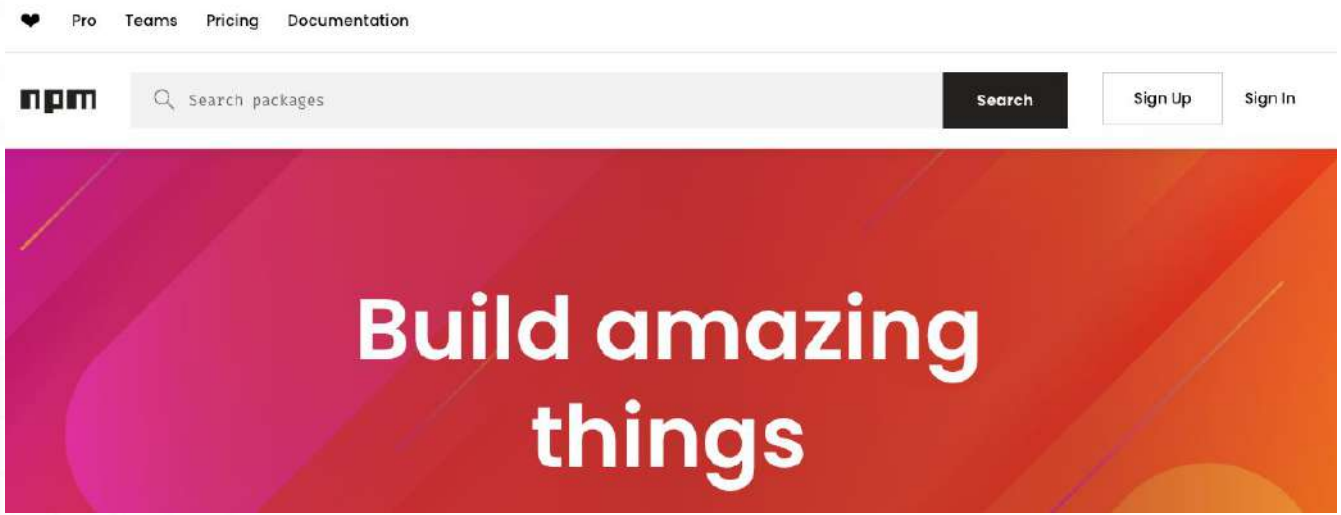
Node carga un módulo una sola vez. En el primer require.

**Las siguientes llamadas a require con la misma ruta devolverán el mismo objeto que se creó en la primera llamada.**

Esto nos vendrá muy bien con las conexiones a bases de datos, por ejemplo.

# JS Pro
# Modulos

El principal repositorio de módulos es:



keep coding

# JS Pro
# Modulos

Los módulos de **EcmaScript** están disponibles de manera estable en Node.js desde hace algunas versiones.

El proyecto debe ser de tipo módulo.

https://nodejs.org/api/packages.html#determining-module-system

keep coding

# JS Pro
# Modulos - ESM

En proyectos con "type": "module", o en ficheros .mjs

- No está disponible require —> se usa import

- No está disponible __dirname —> puedes usar import.meta.dirname*

- No está disponible __filename —> puedes usar import.meta.filename*

keep coding

# HTTP

# Http
## Definición

HTTP es un protocolo de comunicación para transmitir documentos como HTML entre otros.

Es el lenguaje que se utiliza en la web para comunicarse.

Sigue el esquema tradicional de cliente-servidor.

https://developer.mozilla.org/en-US/docs/Web/HTTP

keep coding

# Http
# Methods

Existe un esquema de métodos para definir la **intención** del cliente.

| | | |
|---|---|---|
| - GET | Obtener datos o recursos | Ver listado de usuarios |
| - POST | Enviar datos para crear | Registrar un usuario |
| - PUT | Reemplazar un recurso | Actualizar perfil |
| - PATCH | Modificar parcialmente | Cambiar solo email |
| - DELETE | Eliminar un recurso | Borrar usuario |

En el navegador (a través de la barra de navegación) solo usamos **GET**, los demás métodos se ejecutan de manera programática.

https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Methods

keep coding

# Http
# Status Codes

Http utiliza un sistema de códigos de estado para indicar el resultado de la petición que se deben implementar correctamente.

| | | |
|---|---|---|
| - 1xx | 100-199 | Información (raros de ver). |
| - 2xx | 200-299 | Éxito |
| - 3xx | 300-399 | Redirecciones |
| - 4xx | 400-499 | Errores de cliente (petición incorrecta). |
| - 5xx | 500-599 | Errores de servidor |

https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Status

# Express.js

# Talk is cheap

Show me the code

# Express

Express es un framework web para Node.js

https://expressjs.com/

keep coding

# Express
# Alternativas

Existen múltiples frameworks web para node.js y surgen nuevos con frecuencia, por ejemplo:

- Express.js
- Koa
- Hapi
- Restify
- NestJS
- …

Muchos de ellos extienden la funcionalidad de Express, siendo el más usado.

# Express
# Documentación

Podemos revisar toda la documentación en:

https://expressjs.com/en/api.html

# Express.js

## Express Generator

# Express
## Express Generator

Express Generator nos crea una estructura **base** para nuestra aplicación.

```
$ npx express-generator <nombre-app>
```

keep coding

# Express.js

## Middlewares

keep coding

# Express
# Middlewares

Un middleware es una función que se activa ante una determinada petición, o ante todas.

Controlan el ciclo de vida de una petición y nos permiten conectar diferentes piezas de software entre ellas.

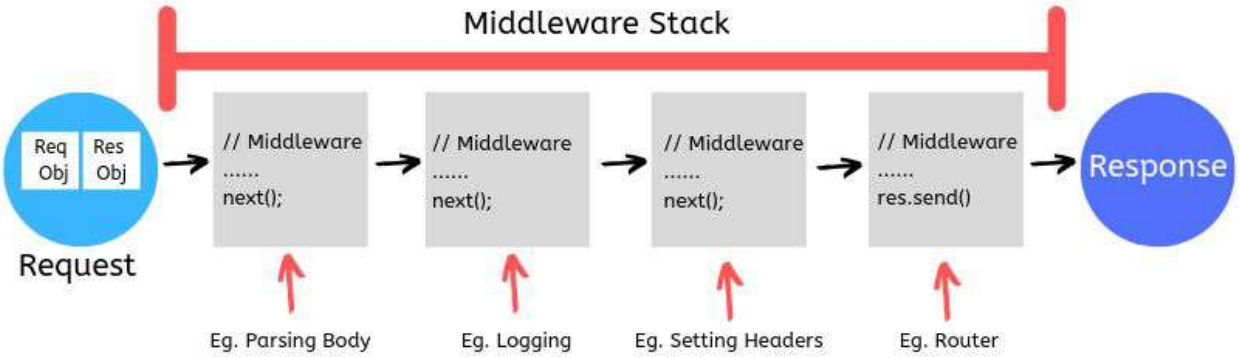
Los middlewares deben **responder** o llamar a `next()`.

Podemos poner tantos middlewares como necesitemos.

El sistema de middlewares de express utiliza callbacks por defecto.

keep coding

# Express
# Middlewares



keep coding

# Express
# Middlewares

```
1  app.use((request, response, next) => {
2      // request contiene toda la información de la petición
3      // response es el objeto que podemos utilizar para contestar
4      // la función next() nos permite continuar con el ciclo de vida
5      console.log('Hola! soy un middleware');
6      next();
7  })
```

# Talk is cheap

Show me the code

keep coding

# Express
# Middlewares - Error

```
1  // Middleware como error handler
2  app.use((error, request, response, next) => {
3      // En una ruta anterior podemos haber hecho next(new Error("Error"));
4      console.error(error.stack);
5      // Podemos devolver lo que más nos convenga
6      res.status(500).send("Internal server error");
7  });
```

keep coding

# Express.js

## Routes

keep coding

# Express
# Routes

En express, una ruta define el comportamiento del servidor cuándo recibe una petición **HTTP** a una dirección (URL) concreta.
https://expressjs.com/en/guide/routing.html

*Una ruta es la combinación de método HTTP + URL + función.*

```
1  app.get('/health', (req, res, next) => {
2      res.status(200).send('OK - healthy');
3  });
```

keep coding

# Express
# Routes

En express, las rutas se evalúan en el orden en que están declaradas. Si dos rutas coinciden, la primera que cumpla la condición será el *handler* de la petición.

```
1  app.get('/', (req, res, next) => { res.status(200).send('Primera') });
2  app.get('/', (req, res, next) => { res.status(200).send('Segunda') });
```

keep coding

# Express
# Routes

Tenemos que interpretar las rutas como condiciones. Cada ruta que declaramos se comporta como un *“if”* que si cumple las condiciones, se encargará de gestionar nuestra petición.

- El método es `${method}`?
- La URL es `${path}`?

keep coding

# Express
# Routes o Middlewares

Las rutas son un tipo de middleware especial que aplica más condiciones.

WTF??

*Spoiler, pueden trabajar en conjunto

```
app.get('/users',
    (req, res, next) => {
        console.log('Middleware para /users');
        next();
    },
    (req, res) => {
        res.json([
            {id: 1, name: 'Alice'},
            {id: 2, name: 'Bob'},
            {id: 3, name: 'Charlie'}
        ]);
});
```

# Talk is cheap

Show me the code

keep coding

# Express.js

## Statics

keep coding

# Express
# Statics

En express, podemos servir ficheros estáticos y funcionar como cualquier servidor web tradicional.

El propio servidor se encarga de devolver los archivos como CSS, imágenes, HTML, javascript, etc.

```
1  app.use( express.static('public') );
```

https://expressjs.com/en/starter/static-files.html

# Talk is cheap

Show me the code

keep coding

# Express.js

## Template engine

# Express
## Template engines

Un motor de plantillas permite generar HTML dinámico en el servidor antes de enviarlo a los navegadores clientes.

En lugar de escribir el HTML a mano dentro de res.send().

# Express
## Template engines

Existen muchos motores de plantillas para Javascript y express soporta la mayoría.

Los más conocidos son:

- PUG
- HBS
- EJS

# Express
## Template engines - EJS

Para empezar a trabajar con un motor de plantillas:

```
$ npm i ejs
```

```
1  app.set('view engine', 'ejs');
2  app.engine('html', renderFile);
3  app.set('views', './views');
```

https://ejs.co/#docs

keep coding

# Gracias

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




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

keep coding

# Descanso

## Buscar articulo domain events

# Descanso

Volvemos 21:19