

Desarrollo Backend

Bienvenid@s

Principios del lenguaje JavaScript I

Clase 02



Pon a grabar la clase



Temario

- Principios del lenguaje JS
 - lo que no está en JS del lado del servidor
- nuestra primera aplicación backend
- variables y constantes (tipos, declaración, asignación)
- Scope (ámbito de una variable)
- funciones (comunes, anónimas, flecha)
- Condicionales
 - if - else
 - switch
 - operador ternario
 - operadores lógicos (AND - OR)



Principios del lenguaje JS

Principios del lenguaje JS

Node JS es el entorno de ejecución que utiliza a JS para poder “*entender*” el código de toda aplicación creada del lado del servidor (*backend*).

Para ello, portó el [Motor V8](#), el cual le permite al entorno Node JS, poder interpretar el código de cada aplicación y compilar el mismo al lenguaje máquina, o binario.



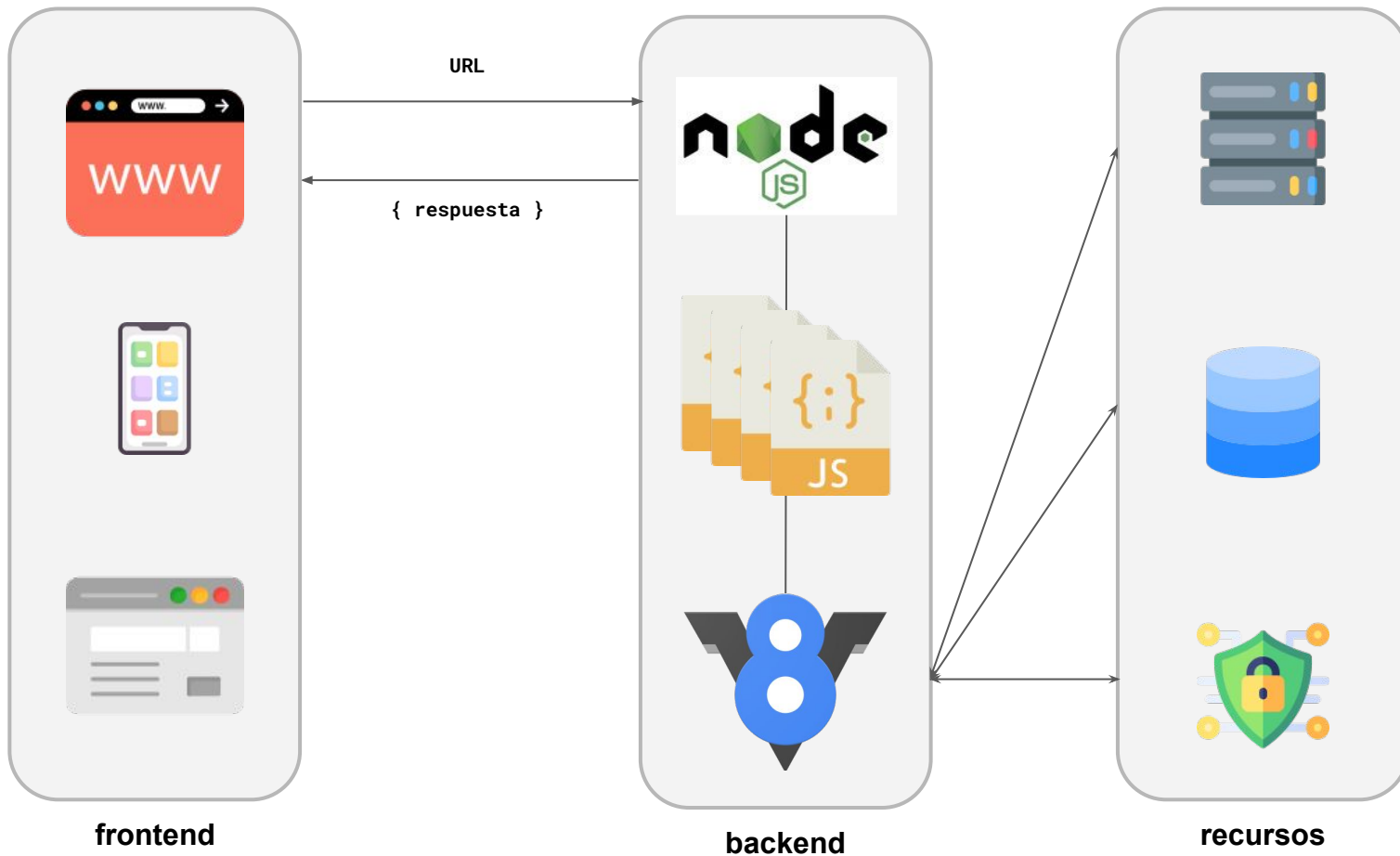
Principios del lenguaje JS

De esta forma, Node JS se desentiende de tener que contar con un compilador, o intérprete, el cual pueda entender el código generado con JavaScript, y traducirlo al lenguaje apropiado que entiende un sistema operativo, un motor de base de datos, un sistema de validación y seguridad, etcétera.



Estos últimos, son los diferentes “*componentes*” que conforman la estructura de la mayoría de las aplicaciones de servidor.

Principios del lenguaje JS



Principios del lenguaje JS

Como desarrolladores de aplicaciones, no debemos comprender todo esto a la perfección, pero sí debemos tener presente una Visión Macro de todo lo que ocurre entre las partes involucradas.

Esto nos ayudará a ser mejores desarrolladores y detectar, de forma efectiva, cualquier error atípico o inesperado que ocurra dentro de todo este proceso técnico.

lo que no está en JS del
lado del servidor

lo que no está en JS del lado del servidor


Si bien el Motor V8 sabe interpretar las instrucciones del lenguaje JavaScript, como el uso de este lenguaje está pensado para ejecutarse en un servidor, como proceso y no como una aplicación frontend, existen algunas limitaciones que debemos tener presente.



Estas limitaciones pertenecen al lenguaje JS, y son instrucciones propias de este que no están presentes cuando JS se ejecuta del lado del servidor. Veamos a continuación cuáles son:

lo que no está en JS del lado del servidor

DOM: el objeto JS **document** y sus métodos asociados para manejar interacción entre JS y HTML y/o CSS, no está disponible del lado del servidor.



```
JavaScript Document

document.querySelector("p.blue-text.grey-bckg");

element.style.fontSize = '26px';

document.createElement('div');
```

lo que no está en JS del lado del servidor

Cuadros de diálogo: `alert`, `prompt` y `confirm`, son cuadros de diálogo nativos de JS que se utilizan en el desarrollo frontend para interactuar con un usuario.

Estas herramientas tampoco las tenemos disponibles en JS del lado del servidor.

```
JavaScript Dialogs

const respuesta = prompt("Ingresa tu nombre de usuario");

const acepto = confirm("¿Desea guardar su nombre de usuario?");

if (acepto) {
    alert(`Su nombre '${respuesta}' ha sido registrado.`);
}
```

lo que no está en JS del lado del servidor

El objeto console: `log()`, `warn()` y `error()`, son mensajes en la consola que denotan diferentes tonos de información. Se usan para depurar de forma correcta en JS. En JS del lado del servidor, se suelen ver todos con la impronta del método `.log()`.

```
> console.log("Esto es un mensaje común");
Esto es un mensaje común
<
> console.warn("Esto es un mensaje de advertencia");
⚠ ▶ Esto es un mensaje de advertencia
<
> console.error("Esto es un mensaje de error! 🤯");
❌ ▶ Esto es un mensaje de error! 🤯
<
```

lo que no está en JS del lado del servidor

Dado que la mayoría de estos objetos, funciones y métodos de JS están pensados para trabajar del lado del usuario (frontend), no los encontraremos disponibles para utilizar al momento de desarrollar aplicaciones de servidor.

Si ya programas aplicaciones web, seguramente tengas presente haber interactuado reiteradas veces con ellos.

lo que ~~no~~ está en JS del lado del servidor

El objeto console: `table()`, es un método del objeto console para estructurar la información de array de elementos y array de objetos.

Este sí está disponible en Node JS para utilizarse, aunque cuando el contenido del array es demasiado, suele romperse la estructura de tabla.

```
13  const frutas = ['Banana', 'Manzana', 'Pera']
14  console.table(frutas)
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

```
[nodemon] restarting due to changes...
[nodemon] starting `node index.js`
```

(index)	Values
0	'Banana'
1	'Manzana'
2	'Pera'

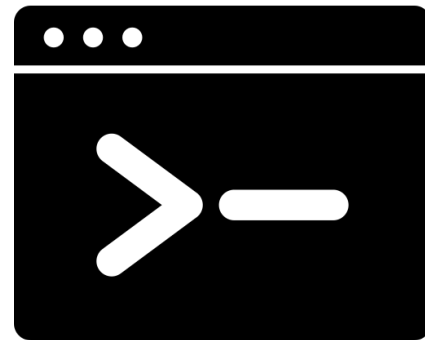
Nuestra primera aplicación backend

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

Nuestra primera aplicación backend

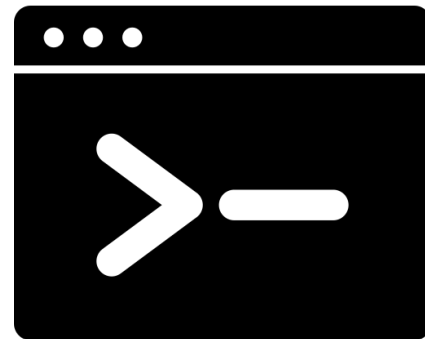
Creemos a continuación nuestra primera aplicación backend. La misma será el clásico *“Hola, Mundo!”*, simple pero conciso, pero nos dará todas las herramientas necesarias para entender cómo manejarnos con JS, la ventana Terminal, y Node JS, en simultáneo.



Nuestra primera aplicación backend

Para ello, crea una carpeta llamada **helloWorld** en el espacio de tu computadora donde almacenas los proyectos de programación.

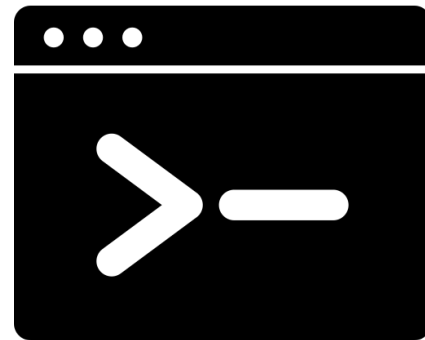
Esto te facilitará tener todo concentrado en un solo lugar, y luego poder resguardarlo en tu repositorio **Github**.



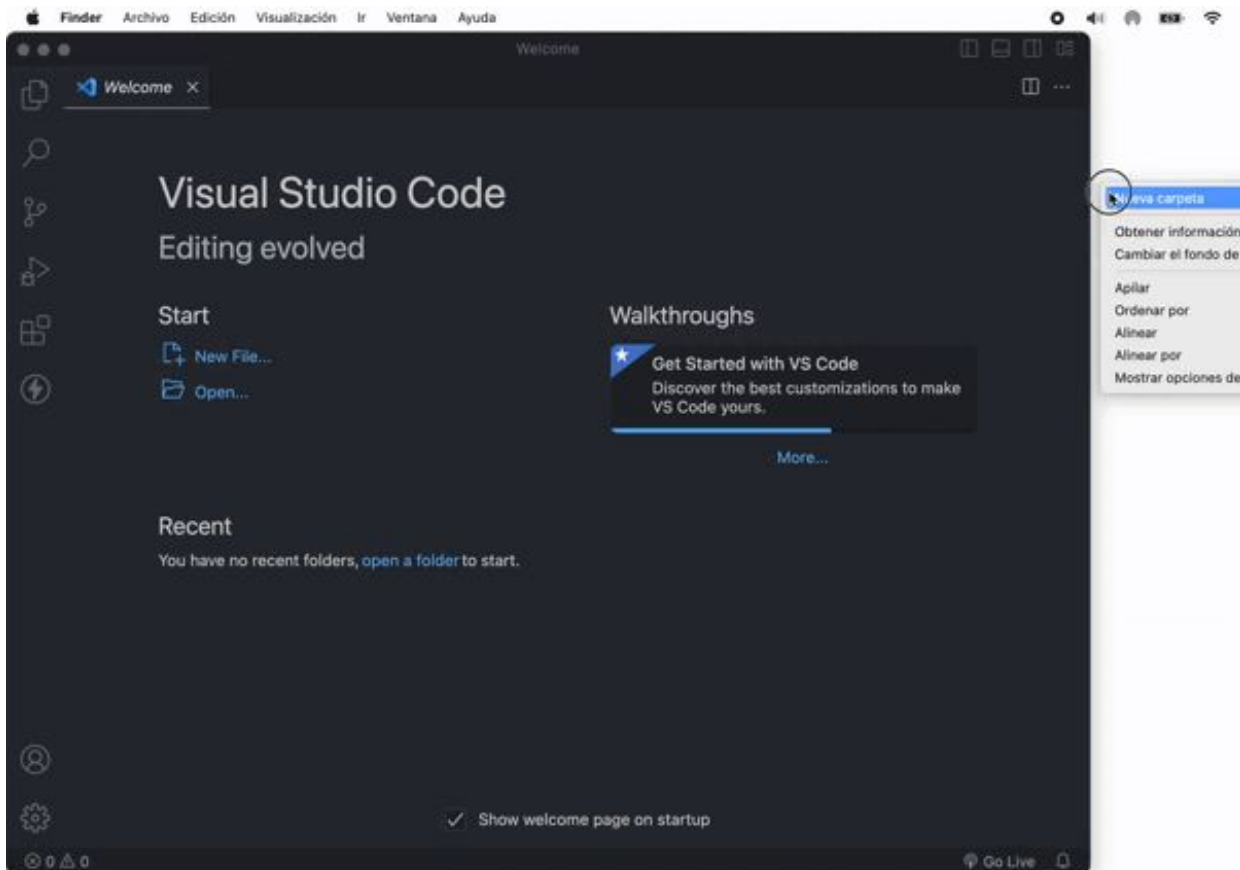
Nuestra primera aplicación backend

Creada la carpeta, abre **VS Code**, y arrastra la misma hacia este editor de código. VS Code abrirá la carpeta como un proyecto nuevo.

Dentro de ésta, ya puedes crear tu primer archivo **.js**. Utiliza el nombre **app.js** al momento de crear el mismo.



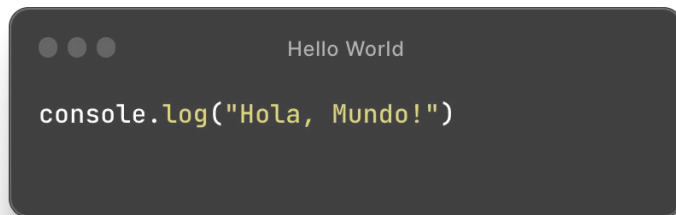
Nuestra primera aplicación backend



Como vemos en esta imagen, el proceso de creación de un proyecto es simple y rápido.

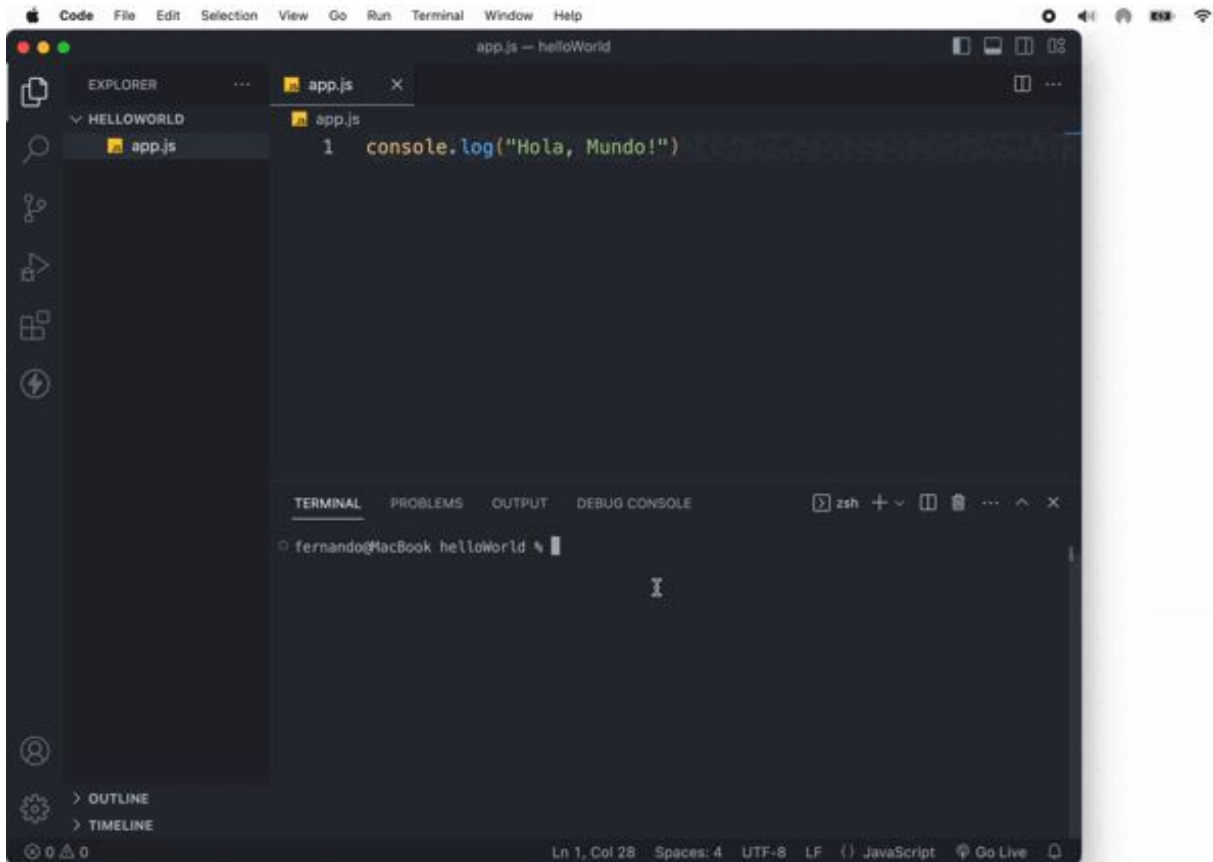
Nuestra primera aplicación backend

Dentro del archivo **app.js**, escribamos el clásico saludo “*Hola, Mundo!*” utilizando **console.log()**.



Seguido a ello, desde el menú **Terminal**, seleccionemos la opción **New Terminal**, para ejecutar nuestra aplicación utilizando Node JS.

Nuestra primera aplicación backend



En la línea de comandos de la ventana Terminal escribimos: **> node app.js**

pulsamos **Enter** y se ejecutará nuestra primera aplicación backend.

Ya tenemos garantía de que nuestro entorno de trabajo funciona correctamente.

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

nuestra primera aplicación backend

Cada vez que necesitemos ejecutar nuestro proyecto, desde la ventana Terminal debemos realizar este paso en particular.

El nombre de archivo `app.js` es opcional. Cualquiera podrá ser el nombre del mismo, aunque siempre debemos respetar las convenciones establecidas en cuanto a claridad técnica refiere.

Repaso del ecosistema JS

Repaso del ecosistema JS

Para poder estar todos en sintonía, realizaremos un repaso por todas las características y sintaxis del lenguaje JavaScript.

Esto servirá para refrescar conocimientos, para actualizar algunos conceptos, y para que todos aquellos que provienen de otro lenguaje de programación, puedan ponerse rápidamente a tono con JS.

¡Manos a la obra!



UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

variables

1- Variables

La declaración de variables se realiza a través de la palabra reservada **let**. También tenemos disponible el uso de la palabra reservada **var**, pero desde la versión 2015 de EcmaScript, la misma se desestima para la implementación de código JS moderno.



Hello World

```
let nombreCompleto = "Joe McMillian";  
let ocupacion = "Founder";  
let empresa = "McMillian Security";
```

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

1- Variables

Los tipos de datos que más utilizamos al crear variables, son:

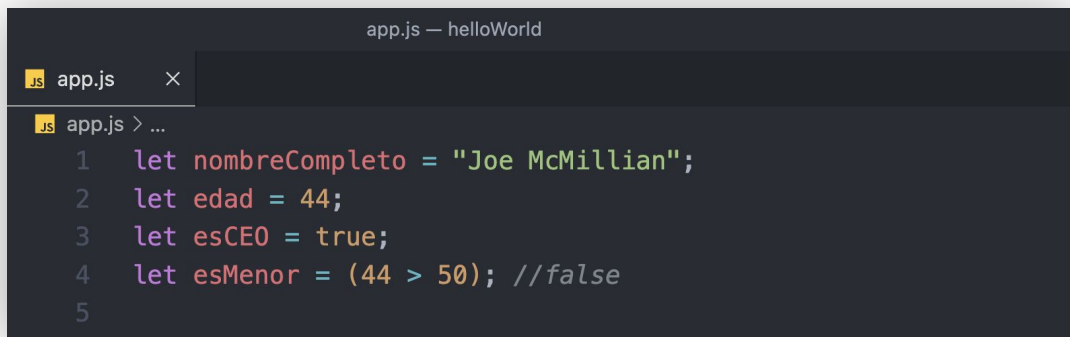
- string (*cadenas de texto*)
- number (*enteros o con decimales*)
- boolean (*verdadero / falso*)

Variables JS

```
let nombreCompleto = "Joe McMillian";  
let edad = 44;  
  
let esCEO = true;  
let esMenor = (44 > 50); //false
```

1- Variables

Si bien, no siempre puede darse esta situación, cuando debemos declarar variables globales se recomienda, (*en lo posible*), definir las al inicio de cada **archivo.js**.

A screenshot of a code editor window titled 'app.js — helloWorld'. The editor shows a file named 'app.js' with a close button. The code inside is as follows:

```
1 let nombreCompleto = "Joe McMillian";
2 let edad = 44;
3 let esCEO = true;
4 let esMenor = (44 > 50); //false
5
```

Esto nos permite ubicar rápidamente las variables, y no “*tener que bucear*” por las líneas de código hasta dar con la misma.

1- Variables

Si bien no siempre puede darse esta situación, cuando debamos declarar variables globales, se recomienda (*en lo posible*), definir a todas ellas al inicio de cada **archivo.js**.

A screenshot of a code editor window titled 'app.js — helloWorld'. The editor shows a file named 'app.js' with a close button. The code inside is as follows:

```
1 let nombreCompleto = "Joe McMillian";
2 let edad = 44;
3 let esCEO = true;
4 let esMenor = (44 > 50); //false
5
```

Esto nos permite ir a buscar rápidamente las variables a su ubicación lógica, y no “*tener que bucear*” por líneas y líneas de código hasta dar con la misma.

1- Variables

Las constantes también son un tipo de variable, pero tal como su nombre lo indica, su valor no puede cambiar *“es constante a lo largo del ciclo de vida de la aplicación”*.

constantes JS

```
const empresa = 'CARDIFF COMPUTERS';  
  
console.log(empresa);
```

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

¿Cuándo usar variables y cuándo usar constantes?

Todo aquel valor que puede o requiere cambiarse en algún momento debe ser declarado como una *variable*.

Aquellos valores que deben ser estáticos o inalterables, debemos declararlos como *constante*.

Ejemplo: {*números de serie, funciones flecha o anónimas, instancias de objetos, exportación de módulos, etcétera...*}

scope

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

2- Scope

```
Scope JS

let nombreCompleto = "Joe McMillian";

console.log("1- Me llamo: ", nombreCompleto);

function mostrarMiNombre() {
  let nombreCompleto = "Gordon Clark";
  console.log("2- Me llamo: ", nombreCompleto);
}

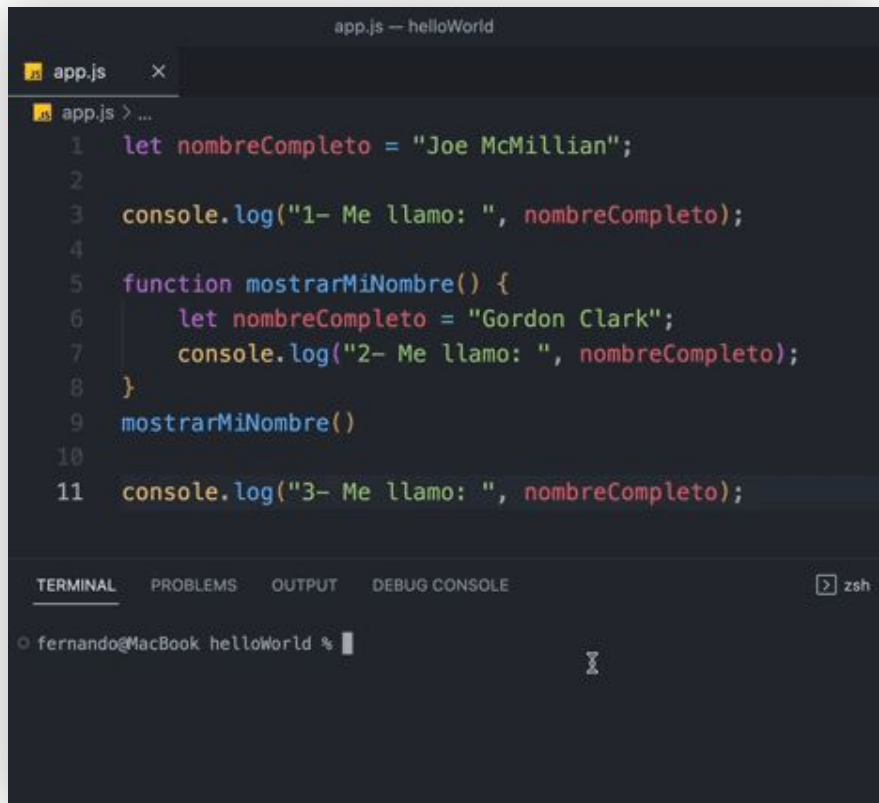
mostrarMiNombre();

console.log("3- Me llamo: ", nombreCompleto);
```

Se define como **Scope**, al alcance o ámbito de una variable.

Las variables JS declaradas con **let** de forma global, pueden ser reutilizadas dentro de una o más funciones, declarándolas nuevamente con otro valor, **sin alterar el valor original global** de la misma.

2- Scope



```
app.js — helloWorld
app.js
1  let nombreCompleto = "Joe McMillian";
2
3  console.log("1- Me llamo: ", nombreCompleto);
4
5  function mostrarMiNombre() {
6      let nombreCompleto = "Gordon Clark";
7      console.log("2- Me llamo: ", nombreCompleto);
8  }
9  mostrarMiNombre()
10
11 console.log("3- Me llamo: ", nombreCompleto);
```

TERMINAL PROBLEMS OUTPUT DEBUG CONSOLE

fernando@MacBook helloWorld %

Aquí vemos en acción el ejemplo de código anterior, utilizando la variable **nombreCompleto**, la cual está declarada como variable global y, a su vez, declarada nuevamente dentro de una función.

Esto no es posible de realizar si utilizamos la palabra reservada **var**.

funciones

3- funciones

```
funciones JS

let nombreCompleto = "Joe McMillian";

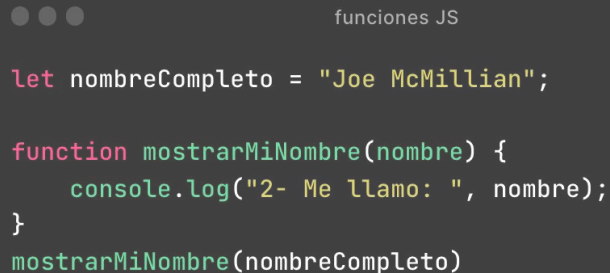
function mostrarMiNombre() {
    console.log("2- Me llamo: ", nombreCompleto);
}

mostrarMiNombre()
```

Las funciones se utilizan exactamente igual que cuando creamos aplicaciones web con JavaScript.

Se estructuran con la palabra reservada **function**, y luego son invocadas en la parte de nuestra aplicación que necesite utilizarlas.

3- funciones con parámetro(s)



```
funciones JS

let nombreCompleto = "Joe McMillian";

function mostrarMiNombre(nombre) {
  console.log("2- Me llamo: ", nombre);
}

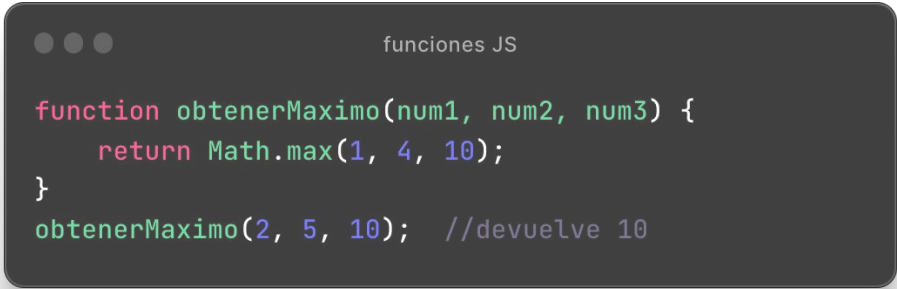
mostrarMiNombre(nombreCompleto)
```

Pueden recibir uno o más parámetros.

Si son más de un parámetro, se deben separar por una coma uno de otros.

El resultado siempre es el mismo.

3- funciones con parámetro(s) y retorno



```
function obtenerMaximo(num1, num2, num3) {  
    return Math.max(1, 4, 10);  
}  
obtenerMaximo(2, 5, 10); //devuelve 10
```

También podemos utilizar las funciones con retorno y/o parámetros.

Para ello, la palabra reservada **return**, será nuestra aliada.

funciones anónimas

3- funciones anónimas

Las funciones anónimas están disponibles también en JavaScript backend.

Su estructura es similar a la declaración de una variable o constante, utilizando seguida a esta la palabra reservada **function**.

funciones anónimas JS

```
const obtenerMaximo = function() {  
  console.log(Math.max(1, 4, 10));  
}  
obtenerMaximo(); //devuelve 10
```

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

3- funciones anónimas

```
funciones anónimas JS

const obtenerMaximo = function(num1, num2, num3) {
  console.log(Math.max(num1, num2, num3));
}

obtenerMaximo(3, 5, 10); //devuelve 10
```

De igual forma, si queremos integrar uno o más parámetros a las funciones anónimas, también podemos realizar esto de la forma convencional.

El resultado en el uso, siempre será el mismo.

3- funciones anónimas

Y, por supuesto, también podemos sumar funciones anónimas con parámetro(s) y retorno de valores.

El uso de todas estas opciones siempre está condicionado a aquellas situaciones donde amerite implementar estos mecanismos de lógica.

funciones anónimas JS

```
const obtenerMaximo = function(num1, num2, num3) {  
  return Math.max(num1, num2, num3);  
}  
obtenerMaximo(3, 5, 10); //devuelve 10
```

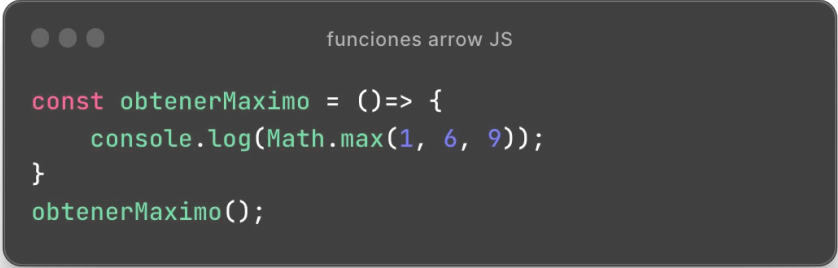
UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

funciones flecha

(arrow functions)

3- funciones flecha



```
funciones arrow JS

const obtenerMaximo = ()=> {
  console.log(Math.max(1, 6, 9));
}

obtenerMaximo();
```

Desde el año 2015, con la llegada de **ES6**, JavaScript sumó las funciones flecha, o arrow functions.

Esta es una estructura moderna de seguir utilizando funciones, pero con algunas ventajas adicionales que simplifican nuestro código.

3- funciones flecha

```
funciones arrow JS

const arrayNumeros = [2, 5, 6, 18, 78, 9, 25];

const obtenerMaximo = array => {
  console.log(Math.max(...array));
}

obtenerMaximo(arrayNumeros);
```

Cuando creamos una **arrow function** con un solo parámetro, podemos prescindir de definir los paréntesis en la estructura de la función.

Luego, cuando llamemos a dicha función, sí debemos declararlos.

3- funciones flecha

```
funciones arrow JS

const arrayNumeros = [2, 5, 6, 18, 78, 9, 25];

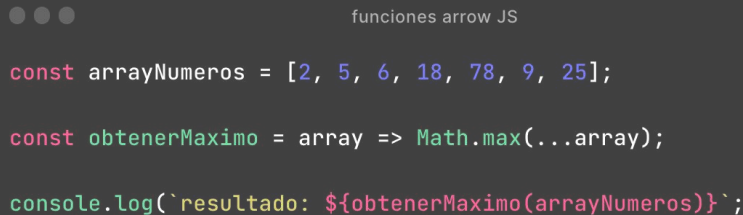
const obtenerMaximo = array => {
  return Math.max(...array);
}

console.log(`resultado: ${obtenerMaximo(arrayNumeros)}`);
```

Las **arrow function** también soportan retorno de resultados, aunque tenemos algunas novedades en cuanto a optimización de este código...

3- funciones flecha

En aquellos casos donde tenemos que retornar un resultado, y la operación contenida dentro de la Arrow Function puede resolverse en una sola línea de código, podemos prescindir de la palabra reservada **return**, y de usar llaves **{ }** contenedoras.



```
funciones arrow JS

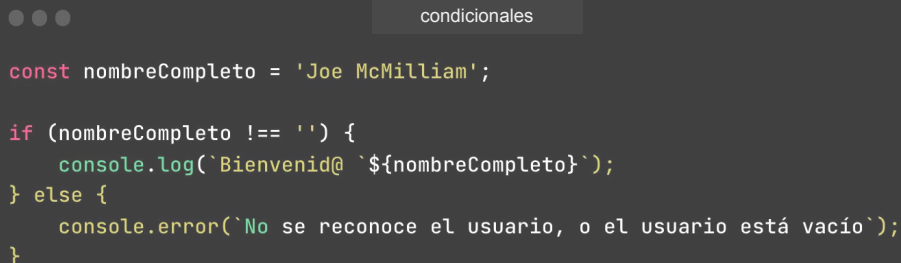
const arrayNumeros = [2, 5, 6, 18, 78, 9, 25];

const obtenerMaximo = array => Math.max(...array);

console.log(`resultado: ${obtenerMaximo(arrayNumeros)}`);
```


Condicionales

4- condicionales



```
const nombreCompleto = 'Joe McMilliam';

if (nombreCompleto !== '') {
  console.log(`Bienvenid@ `${nombreCompleto}``);
} else {
  console.error(`No se reconoce el usuario, o el usuario está vacío`);
}
```

Los condicionales son básicamente aquellas expresiones que nos permiten ejecutar una línea de código en base al resultado del análisis de una expresión.

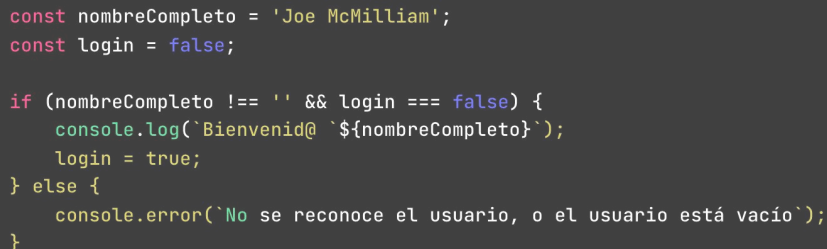
if - else son los condicionales más utilizados en JS, en casi todas las situaciones y lógica de código.

4- condicionales

Podemos utilizar operadores lógicos (**AND** y **OR**), dentro del análisis de una o más expresiones.

Si utilizamos **AND** (**&&**), ambas expresiones deben devolver como resultado TRUE.

Si utilizamos **OR** (**||**), una de las dos expresiones evaluadas debe devolver como resultado TRUE.



```
const nombreCompleto = 'Joe McMilliam';
const login = false;

if (nombreCompleto !== '' && login === false) {
  console.log(`Bienvenid@ `${nombreCompleto}``);
  login = true;
} else {
  console.error(`No se reconoce el usuario, o el usuario está vacío`);
}
```

4- condicionales

a	b	$c = f(a, b)$
V	V	V
V	F	F
F	V	V
F	F	V

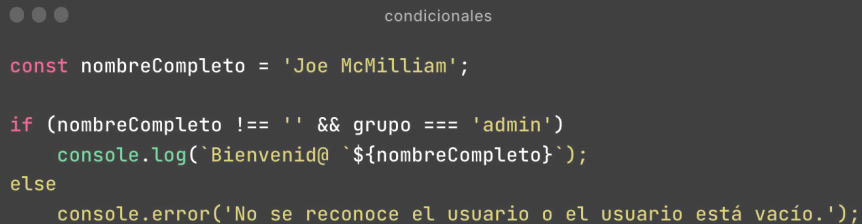
Para este último escenario, **tengamos siempre presente [la Tabla de la Verdad](#)**. La misma es originaria del mundo de las matemáticas, y se aplica con la misma lógica en el mundo de la programación.

Es muy efectiva cuando de evaluar múltiples expresiones se trata.

4- condicionales

JS moderno también permite simplificar la estructura condicional **if**, **if - else**.

Si la misma resuelve la acción precedida por la condición en una sola línea de código, también se puede prescindir del uso de las llaves de bloque.



```
const nombreCompleto = 'Joe McMilliam';

if (nombreCompleto !== '' && grupo === 'admin')
  console.log(`Bienvenid@ `${nombreCompleto}``);
else
  console.error('No se reconoce el usuario o el usuario está vacío.');
```

Operador ternario

5- Operador ternario

```
condicionales

const nombreCompleto = 'Joe McMilliam';

if (nombreCompleto !== '' && grupo === 'admin')
  console.log(`Bienvenid@ `${nombreCompleto}`);
else
  console.error('No se reconoce el usuario o el usuario está vacío.');
```



```
Operador ternario

const nombreCompleto = 'Joe McMilliam';

nombreCompleto !== '' ? console.log(`Bienvenid@ `${nombreCompleto}`);
                        : console.error('No se reconoce el usuario');
```

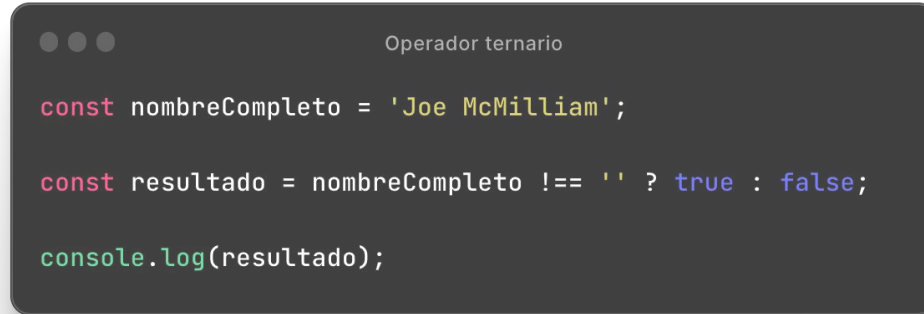
El operador ternario es una evolución simple del **if - else**.

Si el código resultante del bloque if se resuelve en una línea, y de igual forma el código resultante del bloque else, entonces podemos simplificar la estructura utilizando el operador ternario.

? = if

: = else

5- Operador ternario



```
Operador ternario

const nombreCompleto = 'Joe McMilliam';

const resultado = nombreCompleto !== '' ? true : false;

console.log(resultado);
```

El operador ternario funciona con retorno implícito, es decir, el código a ejecutar resultante de la expresión evaluada puede retornar el resultado, y el mismo podemos capturarlo en una variable o constante.

El operador Switch

6- El operador Switch

Switch nos permite evaluar una expresión, pudiendo compararla con múltiples posibles condiciones.

Cuando se cumple la condición, ejecutamos el código asociado y cortamos la comparación utilizando **break**.

Si ninguna se cumple, podemos definir un mensaje estándar de la mano de **default**.



```
let color = 'Rojo';

switch (color) {
  case 'Blanco':
    console.warn('No es el color esperado');
    break;
  case 'Azul':
    console.warn('No es el color esperado');
    break;
  case 'Rojo':
    console.log('Este es el color que buscaba.');
```

6- El operador Switch

```
let color = 'Rojo';

switch (color) {
  case 'Blanco':
    console.warn('No es el color esperado');
    break;
  case 'Azul':
    console.warn('No es el color esperado');
    break;
  case 'Rojo':
    console.log('Este es el color que buscaba.');
```

Operador ternario

Siempre debemos usar la cláusula **break** para interrumpir el proceso de análisis de **Switch**.

Podemos prescindir de este en la opción por defecto (**default**).

Como contra, la comparación que requiere hacer Switch no es estimada, sino absoluta.

(no podemos comparar “mayor a”, “menor o igual”, etcétera)

operadores lógicos

(AND - OR)

7- Operadores lógicos AND - OR

```
Operator lógico AND

let color = 'Rojo';

if (color === 'Rojo') {
  console.log('El color elegido es el correcto');
}

//CON EL OPERADOR LÓGICO AND

(color === 'Rojo') && console.log('El color elegido es el correcto');
```

El operador lógico **AND** (&&) puede utilizarse en determinadas ocasiones para simplificar una estructura de control del tipo **if** simple.

Si el código a ejecutar asociado se resuelve en una sola línea, entonces podemos optar por el operador lógico **AND** para simplificar el uso de **if**.

7- Operadores lógicos AND - OR

El operador lógico **OR** (`||`), funciona para detectar un posible cortocircuito ante un valor inesperado.

De esta forma, podemos establecer un valor predeterminado si el valor a recuperar no cumple con algún resultado válido para valores del tipo ***falsy***.

Operador lógico OR

```
obtenerCarritoConProductos(); //devuelve null  
  
const carrito = obtenerCarritoConProductos() || [];
```

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

7- Operadores lógicos AND - OR

Decimos que un valor es del tipo **Falsy**, cuando el resultado esperado no es el efectivo.

Veamos en la tabla contigua, las diferentes representaciones de valores esperados versus los valores recibidos.

Valor esperado	Valor resultante	¿Es falsy?
1234	NaN	Sí
['Banana', 'Manzana']	null	Sí
2103	0	Sí
1407	-0	Sí
true	false	Sí
'Joe McMillian'	undefined	Sí
{profe: 'Gordon Clark'}	{}	Sí

Utilizando el **operador lógico OR**, podemos definir “*ante este cortocircuito*”, un valor predeterminado. Así, el resto de la lógica de JS, podrá reaccionar de manera efectiva.

Espacio de trabajo

Espacio de trabajo

Pongamos en práctica este repaso general de la sintaxis básica y moderna de JavaScript, a través de un conjunto de ejercicios.

Tiempo estimado: 20 minutos. 

UNTREF

UNIVERSIDAD NACIONAL
DE TRES DE FEBRERO

```
const questions = [ 'dudas', 'consultas', '🧐' ]
```



```
> node gracias.js
```